Kubernetes 经典实例

Kubernetes Cookbook

Sébastien Goasguen, Michael Hausenblas 著
马晶慧 译

中国电力出版社

对本书的赞誉

Kubernetes 是史上最好的基础设施。本书可以帮助你学习这个最好的基础设施。这本书提供了能够解决实际问题的具体示例，你可以将其应用到实际工作中。认真阅读本书，你能够学习真正的技术，将自己的 Kubernetes 提升到一个新的水平。

—— Joe Beda
Heptio 的 CTO 兼创始人，
Kubernetes 创始人

本书是一本非常优秀的实用指南，教你如何在 Kubernetes 上建立和运行应用程序。它是帮助你学习如何建立云端原生容器化应用程序很好的参考和方式。

—— Clayton Coleman
Red Hat

在这本书中，本书作者收集了大量实用的技巧，帮助你迅速地掌握 Kubernetes。他们整理了几十个非常实用的提示和技巧，可以帮助读者从实用性的角度掌握如何安装 Kubernetes，以及利用 Kubernetes 运行应用程序。

—— Liz Rice
首席技术传播者，
Aqua Security

Kubernetes经典实例

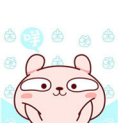
Sébastien Goasguen, Michael Hausenblas 著

马晶慧 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社



图书在版编目（CIP）数据

Kubernetes经典实例 / (美) /塞巴斯蒂安·戈阿冈（Sébastien Goasguen），(美) 迈克尔·豪森布拉斯（Michael Hausenblas）著；马晶慧译. — 北京：中国电力出版社，2018.10

书名原文：Kubernetes Cookbook

ISBN 978-7-5198-2399-3

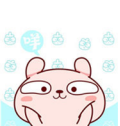
I. ①K… II. ①塞… ②迈… ③马… III. ①Linux操作系统－程序设计 IV. ①TP316.85

中国版本图书馆CIP数据核字(2018)第208161号

北京市版权局著作权合同登记 图字：01-2018-3072号

出版发行：中国电力出版社
地　　址：北京市东城区北京站西街19号（邮政编码100005）
网　　址：http://www.cepp.sgcc.com.cn
责任编辑：刘 炽（liuchi1030@163.com）
责任校对：王小鹏
装帧设计：Randy Comber，张 健
责任印制：蔺义舟

印　刷：北京天宇星印刷厂
版　次：2018年10月第一版
印　次：2018年10月北京第一次印刷
开　本：750毫米×980毫米　16开本
印　张：13.25
字　数：245千字
印　数：0001—3000册
定　价：48.00元



O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

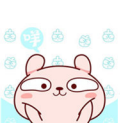
——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal



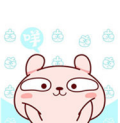
献给我的儿子们，你们的微笑、拥抱和鼓励让我成了一个更好的人。

献给与我共度此生的妻子。

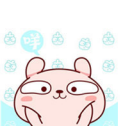
—— Sébastien

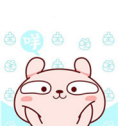
献给Saphira、Ranya、Iannis和Anneliese。

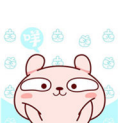
—— Michael

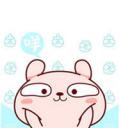


目录









前言

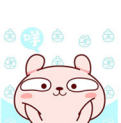
欢迎你阅读 Kubernetes 经典实例，感谢选择我们的书籍！在本书中，我们将帮助你解决关于 Kubernetes 的具体问题。我们总结了 80 多个技巧，主题包括建立集群、通过 Kubernetes API 对象管理容器化 workload、使用存储的基本方法、安全配置以及 Kubernetes 的扩展等。无论你是 Kubernetes 新手，还是有一定的经验，我们希望你都可以从本书中找到有用的信息，提高你的经验和对 Kubernetes 的应用。

本书面向的读者对象

无论你是云端原生开发人员，或是系统管理员，或是这种新型的开发运维人员，本书都可以帮助你成功地在 Kubernetes 丛林中找到出路，帮你从开发走向正式产品。本书中的各个技巧并没有按照 Kubernetes 的基本概念线性展开，但是，每章包含的技巧都会运用 Kubernetes 的核心概念和 API 的基本方法。

为什么编写本书

我们两个人使用 Kubernetes 已经很多年了，并向其贡献了很多代码，我们看到了很多新手甚至是高级用户也会遇到的问题。我们想与大家分享我们积累的知识，包括在产品或在开发环境中运行 Kubernetes 的知识，以及开



发 Kubernetes 的经验，例如向核心代码库或生态系统贡献代码，以及编写在 Kubernetes 上运行的应用程序。

本书的组织结构

本书包括 14 章。每章都是由若干以 O'Reilly 标准的提问模式（问题，解决方案，讨论）编写的小节组成的。你可以从头到尾依次阅读本书，也可以跳过某些章节。每个章节都是独立的，并且在理解一些其他章节的概念的时候，我们也提供了相应的备注。有些章节会展示具体的命令。

关于 Kubernetes 版本发行的说明

写这本书的时候，Kubernetes 1.7 是最新的稳定版本，于 2017 年 6 月末发行，也是我们在本书中使用的版本[注 1]。然而，书中所展示的解决方案普遍适用于其他旧版本，至少到 Kubernetes 1.4 都没问题。如果解决方案只适用于新版本的话，我们会做明确的解释，并列出所需的最低版本。

2017 年，Kubernetes 在每个季度都会推出新的版本，例如：3 月的版本 1.6，6 月的版本 1.7，9 月的版本 1.8，到 12 月本书的英文版发布的时候，Kubernetes 推出了版本 1.9。Kubernetes 的版本发行标准明确指出了每个功能都可以在 3 个小版本期间得到支持[注 2]。这意味着版本 1.7 中稳定的 API 对象的支持会一直持续到 2018 年 3 月。然而，因为本书在大多数时候仅使用稳定的 API，所以即便你使用的是更新版本的 Kubernetes，各个章节中提及的解决方案也依然有效。

注 1：请查阅文档："Kubernetes 1.7: Security Hardening, Stateful Application Updates and Extensibility"（*https://kubernetes.io/blog/2017/06/kubernetes-1.7-security-hardening-stateful-application-extensibility-updates*）。

注 2：请查阅文档："Kubernetes API and Release Versioning"（*https://github.com/eBay/Kubernetes/blob/master/docs/design/versioning.md*）。

你需要掌握的技术

这是一本中级水平的书，需要读者对一些开发和系统管理的概念有最基本的理解。在阅读本书前，请先确认对以下技术有基本的理解：

bash（UNIX shell）

bash 是默认的 Linux 和 MacOS 的 UNIX shell。你需要熟悉 UNIX shell 的知识，例如编辑文件，设定文件许可和用户权限，在文件系统内移动文件，以及一些基本的 shell 编程。如果你想学习 bash 的基本知识，请参阅 O'Reilly 出版的 Cameron Newham 的著作《Learning the bash Shell》，或 JP Vossen 和 Carl Albing 的著作《bash Cookbook》。

包管理

本书中提及的工具常常具有依赖性，需要安装一些软件包。因此，你需要了解包管理系统方面的知识。包管理系统可以是 Ubuntu/Debian 系统中的 *apt*，CentOS/RHEL 系统中的 yum，或者 MacOS 的 port 或 brew 命令。无论是哪一种，请确认你了解如何安装、升级或删除软件包。

Git

Git 已成为标准的分布式版本控制工具。如果你熟悉 CVS 和 SVN，但是还未曾用过 Git，那么应该尝试一下。Jon Loeliger 和 Matthew McCullough 合著的《Version Control with Git》（O'Reilly 出版）是个好的开始。同时，GitHub 网站是优秀的资源，可以用于托管个人的代码仓库。更多关于 GitHub 的信息，请访问 *http://training.github.com* 和相关的交互式教程（*https://try.github.io/levels/1/challenges/1*）。

Python

除了 C/C++ 或 Java 之外，我们总是鼓励学生选择一种脚本语言。曾经 Perl 是脚本语言的主宰，不过目前 Ruby 和 Go 似乎更加流行。本书中的大多数例子使用的都是 Python，但是也有几个 Ruby 的例子，还有一个甚至用到了 Clojure。O'Reilly 出版了很多关于 Python 的书籍，包括 Lubanovic 的《Introducing Python》、Mark Lutz 的《Programming Python》，以及 David Beazley 和 Brian K. Jones 合著的《Python Cookbook》。

Go

Kubernetes 是用 Go 写的。过去几年中，Go 已经成为了许多创业公司和系统相关的开源项目的新编程语言的首选。这本书并没有涉及 Go 编程，但是演示了如何编译 Go 项目。所以希望你对如何建立 Go 工作空间有基本的理解。如果想了解更多关于 Go 的知识，可以参阅 O'Reilly 的视频培训教程“Introduction to Go Programming”。

在线资源

Kubernetes 清单文件、代码示例和其他本书中用到的脚本都保存到了 GitHub 上（*https://github.com/k8s-cookbook/recipes*）。你可以拷贝这个代码仓库，然后阅读相应的章节，并使用这些代码：

```
$ git clone https://github.com/k8s-cookbook/recipes
```

这个代码仓库中的示例并不是在生产环境中使用的最佳设置。这些示例代码只是为了运行各个章节中的示例而编写的最简代码。

排版约定

本书使用了下述排版约定。

斜体（Italic）

表示新术语、URL、示例电子邮件地址、文件名和扩展名。

等宽字体（`Constant Width`）

表示代码，在段内用以表示与代码相关的元素，例如变量或函数名、数据库、数据类型、环境变量、声明和关键字。还用于命令和命令的结果输出。

等宽粗体字（**`Constant width bold`**）

表示命令或其他用户输入的文本。

斜体等宽字体（*Constant Width Italic*）

表示该文本应当由用户提供的值或由上下文决定的值。

表示提示或建议。

表示一般性说明。

表示警告或提醒。

使用代码示例

本书的目的是帮助你完成工作。一般来说，你可以在自己的程序或者文档中使用本书附带的示例代码。你无需联系我们获得使用许可，除非你要复制大量的代码。例如，使用本书中的多个代码片段编写程序就无需获得许可。但以 CD-ROM 的形式销售或者分发 O'Reilly 书中的示例代码则需要获得许可。回答问题时援引本书内容以及书中示例代码，无需获得许可。在你自己的项目文档中使用本书大量的示例代码时，则需要获得许可。

我们不强制要求署名，但如果你这么做，我们深表感激。署名一般包括书名、作者、出版社和国际标准图书编号。例如："Kubernetes Cookbook by Sébastien Goasguen and Michael Hausenblas (O'Reilly). Copyright 2018 Sébastien Goasguen and Michael Hausenblas, 978-1-491-97968-6"。

如果你觉得自身情况不在合理使用或上述允许的范围内，请通过邮件和我们联系，地址是 *permissions@oreilly.com*。

O'Reilly Safari

Safari（以前的 Safari Books Online）是面向企业、政府、教育和个人的会员制培训与参考平台。

Safari 的会员可以访问成千上万的书籍、培训视频、学习路径、交互式教程和推荐的书单。这些内容由 250 多家出版社提供，其中包括：O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology 等。

更多关于 Safari 的信息，请访问我们的网站：*http://oreilly.com/safari*。

联系我们

请把你对本书的意见和疑问发给出版社：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

这本书有专属网页，你可以在那里找到本书的勘误、示例和其他信息。这个网页的地址是*http://bit.ly/kubernetes-cookbook*。

如果你对本书有一些评论或技术上的建议，请发送电子邮件到*bookquestions@oreilly.com*。

要了解O'Reilly图书、培训课程、会议和新闻的更多信息，请访问我们的网站，地址是：*http://www.oreilly.com*。

请在 Facebook 上联系我们，地址是：*http://facebook.com/oreilly*。

请在 Twitter 上关注我们，地址是：*http://twitter.com/oreillymedia*。

请观看我们的 Youtube视频：地址是：*http://www.youtube.com/oreillymedia*。

致谢

感谢整个Kubernetes社区开发了如此优秀的软件，我们是一个伟大的大家族，每个人都很开放，很热心，总是乐于助人。

创作这本书的过程比预想的困难，但我们还是完成了这项工作，我们对所有帮助过我们的人表示衷心的感谢。尤其要感谢 Ihor Dvoretski、Liz Rice 和 Ben Hall 的全面评论，他们帮助我们修正了很多问题，并提出了对所有读者都有帮助的更好的组织方式和章节。

第 1 章

初识 Kubernetes

作为开篇，我们将通过本章的各节帮助你初步了解 Kubernetes。我们将展示如何在不安装的情况下使用 Kubernetes，还将介绍一些组件，比如命令行界面和仪表盘，你可以通过它们访问集群，也可以使用 Minikube——一个可以让开发人员在本地使用和运行 Kubernetes 集群的工具。

1.1 无需安装即可使用 Kubernetes

问题

如何在不安装的情况下使用 Kubernetes？

解决方案

可以参考 Kubernetes 网站提供的互动教程（*https://kubernetes.io/docs/tutorials/kubernetes-basics/*），无需安装就可以使用 Kubernetes。

还可以尝试 Katacoda 提供的 Kubernetes 练习环境（*https://www.katacoda.com/courses/kubernetes/playground*）。用 GitHub 或其他社交网络的账号登录该网站后，可以看到如图 1-1 所示的页面。

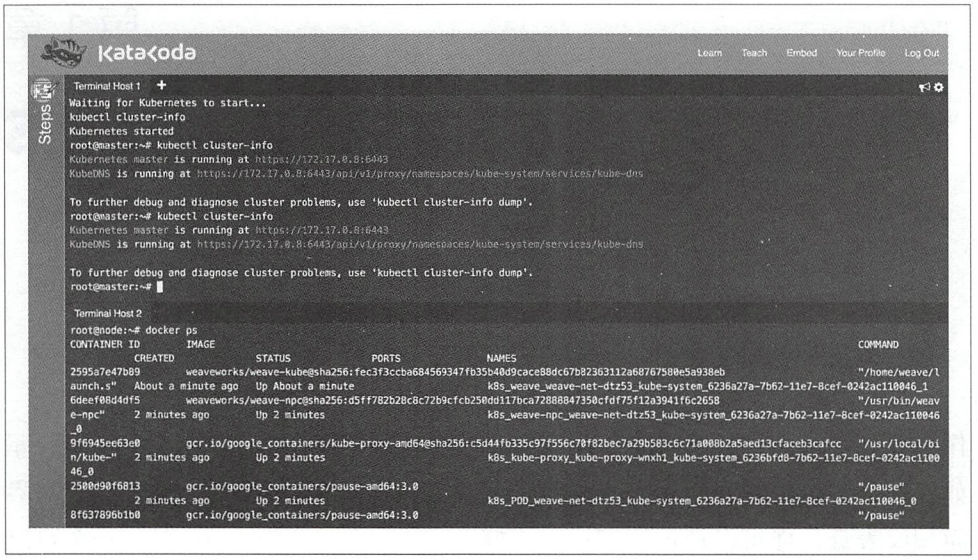


图 1-1：Katacoda 提供的 Kubernetes 练习环境的截图

请注意这个网站的练习环境有时间限制，目前是 1 小时，但这个环境是完全免费的，可以直接通过浏览器访问。

1.2 安装 Kubernetes 的命令行界面和 kubectl

问题

如何安装 Kubernetes 的命令行界面，并通过这个界面与 Kubernetes 集群进行交互？

解决方案

可以通过下面几种方法安装 kubectl：

- 下载源代码的压缩包（.tar 格式）。
- 使用包管理器。
- 从源代码编译（请参阅 13.1 节）。

Kubernetes 的官方文档给出了获取 kubectl 的几种方法（*https://kubernetes.io/docs/tasks/tools/install-kubectl/*）。最简单的方法是下载官方的最新版本。例如，Linux 系统可以通过以下命令获得最新的稳定版本：

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/ \
       $(curl -s https://storage.googleapis.com/kubernetes-release/ \
       release/stable.txt) \
       /bin/linux/amd64/kubectl


$ chmod +x ./kubectl

$ sudo mv ./kubectl /usr/local/bin/kubectl
```

MacOS 的用户可以通过 Homebrew 获取 kubectl：

```
$ brew install kubectl
```

而对于 Google Kubernetes Engine 用户（请参阅 2.7 节），gcloud 的安装命令将同时安装 kubectl。例如，在本书作者本地的计算机上，kubectl 安装在如下位置：

```
$ which kubectl
/Users/sebgoa/google-cloud-sdk/bin/kubectl
```

请注意最新版本的 Minikube（请参阅 1.3 节）包含了 kubectl，如果在安装时找不到 kubectl，它会自动安装到 $PATH 所指的地方。

在本节结束之前，请运行查询 kubectl 版本的命令，以确认它可以正常工作。这个命令还会返回默认的 Kubernetes 集群的版本：

```
$ kubectl version
Client Version: version.Info{Major:"1", \
                             Minor:"7", \
                             GitVersion:"v1.7.0", \
                             GitCommit:"fff5156...", \
                             GitTreeState:"clean", \
                             BuildDate:"2017-03-28T16:36:33Z", \
                             GoVersion:"go1.7.5", \
                             Compiler:"gc", \
                             Platform:"darwin/amd64"}
...
```

请参阅

- 关于安装 kubectl 的官方文档（*https://kubernetes.io/docs/tasks/tools/install-kubectl/*）。

1.3 安装 Minikube 并运行本地的 Kubernetes 实例

问题

如何在本地机器上利用 Kubernetes 进行测试、开发或培训？

解决方案

可以使用 Minikube。Minikube 是一个工具，你只需要在本地机器上安装 Minikube 可执行文件（不需要其他任何软件），就可以使用 Kubernetes。它可以利用本地的虚拟机监视器（比如 VirtualBox、KVM 等）启动虚拟机，并在单一节点上运行 Kubernetes。

可以通过获取最新版本或源代码编译的方式，在本地安装 Minikube 命令行界面。在 Linux 系统上，可以通过如下命令获取并安装 v0.18.0 版本的 Minikube：

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/v0.18.0/ \
                    minikube-linux-amd64

$ chmod +x minikube

$ sudo mv minikube /usr/local/bin
```

上述命令会把 Minikube 可执行文件放到环境变量的 path 上，所以无论在任何地方都可以访问 Minikube。

讨论

在安装了 Minikube 之后，可以通过运行如下命令确认其版本信息：

```
$ minikube version
minikube version: v0.18.0
```

启动 Minikube 的命令为：

```
$ minikube start
```

启动步骤结束后，Kubernetes 的客户端 kubectl 就拥有了 Minikube 环境，它会自动使用该环境。如下检查集群所包含节点的命令可以返回 Minikube 的主机名：

```
$ kubectl get nodes
NAME       STATUS    AGE
minikube   Ready     5d
```

请参阅

- Minikube 的官方文档（*https://kubernetes.io/docs/getting-started-guides/minikube/*）。
- GitHub 上 Minikube 的源代码（*https://github.com/kubernetes/minikube*）。

1.4 在本地使用 Minikube 进行开发

问题

如何使用 Minikube 在本地进行测试和开发 Kubernetes 的应用程序？你已经安装并启动了 Minikube（请参阅 1.3 节），现在想知道一些命令以简化开发过程。

解决方案

Minikube 的命令行界面提供了一些实用的命令。这个命令行界面包含内置的帮助文档，可以用于查看子命令。下面是部分帮助文档的节选：

```
$ minikube
...
Available Commands:
  addons          Modify minikube's kubernetes addons.
...
  start           Starts a local kubernetes cluster.
  status          Gets the status of a local kubernetes cluster.
  stop            Stops a running local kubernetes cluster.
  version         Print the version of minikube.
```

除了 `start`、`stop` 和 `delete`，你还应该对 `ip`、`ssh`、`dashboard` 和 `docker-env` 等命令有所了解。

Minikube 通过运行 Docker 引擎可以启动容器。为了通过本地的 Docker 客户端在本地计算机上访问 Docker 引擎，你需要使用 `minikube` 的 `docker-env` 设置正确的 Docker 环境。

讨论

`minikube start` 命令可以启动虚拟机，以便在本地运行 Kubernetes。默认情况下，它会给虚拟机分配 2GB 的内存，所以结束的时候别忘了使用 `minikube`

stop 命令关闭虚拟机。而且，你可以给虚拟机指定更多内存和 CPU，或运行特定的 Kubernetes 版本，例如：

```
$ minikube start --cpus=4 --memory=4000 --kubernetes-version=v1.7.2
```

为了调试 Minikube 内部使用的 Docker 服务，可以利用 minikube ssh 命令方便地登录虚拟机。minikube ip 命令可以获取 Minikube 虚拟机的 IP 地址。最后，minikube dashboard 命令可以在默认的浏览器中启动 Kubernetes 的仪表盘。

如果 Minikube 突然出现不稳定的状况，需要重启时，可以使用 minikube stop 和 minikube delete 命令删除当前的 Minikube，然后通过 minikube start 命令重新安装。

1.5 在 Minikube 上运行应用程序

问题

你已经启动了 Minikube（请参阅 1.3 节），现在如何在 Kubernetes 上启动第一个应用程序呢？

解决方案

举例来说，你可以使用如下两个 kubectl 的命令启动 Ghost 微型博客平台：

```
$ kubectl run ghost --image=ghost:0.9
$ kubectl expose deployments ghost --port=2368 --type=NodePort
```

使用如下命令可以手动监视该 pod，查看它何时启动，还可以利用 minikube service 命令自动打开浏览器并访问 Ghost：

```
$ kubectl get pods
```

```
NAME                     READY     STATUS    RESTARTS   AGE
ghost-8449997474-kn86m   1/1       Running   0          2h

$ minikube service ghost
```

讨论

`kubectl run` 命令被称作“生成器”，利用这个命令可以很方便地创建 Deployment 对象（请参阅 4.4 节）。`kubectl expose` 命令也是生成器，可以用于创建 Service 对象（请参阅 5.1 节），Service 对象可以提供路由服务，把网络流量分配到部署时启动的容器上。

1.6 使用 Minikube 访问仪表盘

问题

如何使用 Minikube 访问 Kubernetes 的仪表盘，并从图形用户界面上启动第一个应用程序呢？

解决方案

可以从 Minikube 上使用如下命令打开 Kubernetes 的仪表盘：

```
$ minikube dashboard
```

单击浏览器右上角的“+”号，可以看到如图 1-2 所示的页面。

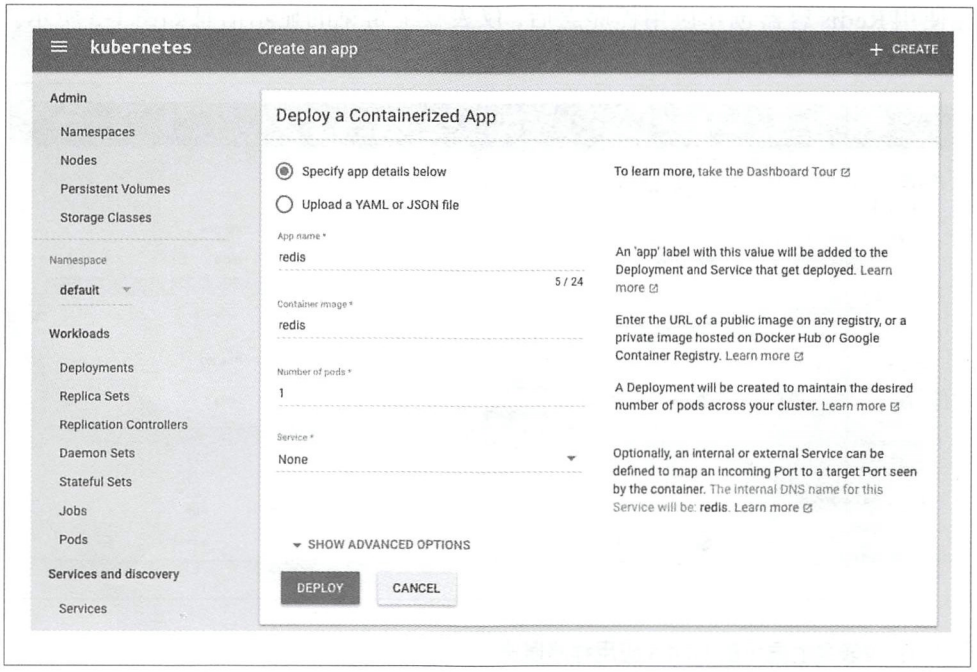


图 1-2：在仪表盘中创建应用程序的截图

讨论

如果想创建应用程序，那么可以单击右上角的“Create”（创建）按钮，填写应用程序的名称，并指定想使用的 Docker 映像。然后单击“Deploy”（部署）按钮，就会看到一个新的页面，上面显示了部署和复制节点的信息，过一段时间后还会看到一个 pod。这些是一些关键的 API 原语，我们会在本书的后续章节详细讨论。

在使用 Redis 容器创建应用程序之后，仪表盘上常见的显示信息如图 1-3 所示。

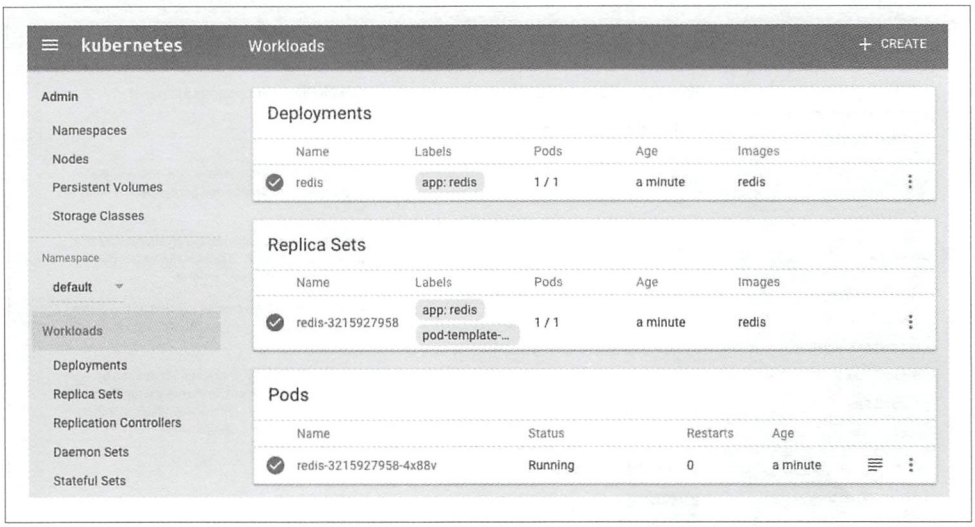


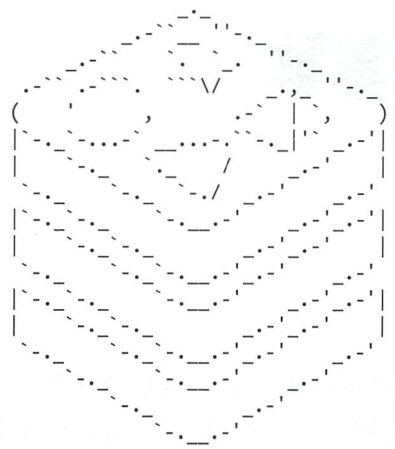
图 1-3：仪表盘上显示的 Redis 应用程序概览

如果从终端上使用命令行客户端，也可以看到相同的信息：

```
$ kubectl get pods,rs,deployments
NAME                          READY     STATUS    RESTARTS   AGE
po/redis-3215927958-4x88v     1/1       Running   0          24m

NAME                   DESIRED   CURRENT   READY     AGE
rs/redis-3215927958    1         1         1         24m

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/redis   1         1         1            1           24m
```

Redis pod 将会运行 Redis 的服务器，可以通过如下日志文件确认：

```
$ kubectl logs redis-3215927958-4x88v
...
```



```
                Redis 3.2.9 (00000000/0) 64 bit

                Running in standalone mode
                Port: 6379
                PID: 1


                      http://redis.io
```

```
...
1:M 14 Jun 07:28:56.637 # Server started, Redis version 3.2.9
1:M 14 Jun 07:28:56.643 * The server is now ready to accept connections on
port 6379
```

第 2 章

创建 Kubernetes 集群

本章我们将介绍建立成熟的 Kubernetes 集群的几种方法。我们会介绍底层标准化的工具 `kubeadm`，这个工具也是其他安装程序的基础，同时我们还会介绍从何处获取控制平面以及工作节点相关的可执行文件。我们还将介绍使用 `hyperkube` 建立容器化的 Kubernetes，并演示如何创建 `systemd` 文件以监管 Kubernetes 组件。本章末尾我们还将介绍如何在 Google Cloud 和 Azure 上建立集群。

2.1 安装 kubeadm 以创建 Kubernetes 集群

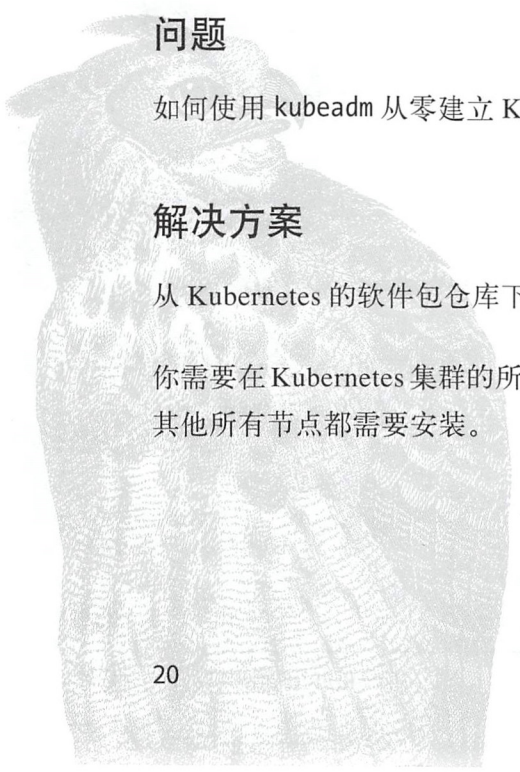
问题

如何使用 `kubeadm` 从零建立 Kubernetes 集群？

解决方案

从 Kubernetes 的软件包仓库下载 `kubeadm` 命令行界面工具。

你需要在 Kubernetes 集群的所有服务器上安装 `kubeadm`。不仅主节点需要安装，其他所有节点都需要安装。



例如，如果使用 Ubuntu 的主机，那么必须在每个主机上以 root 身份运行下面的命令，以设置 Kubernetes 的软件包仓库：

```
# apt-get update && apt-get install -y apt-transport-https

# curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

# cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
  deb http://apt.kubernetes.io/ kubernetes-xenial main
  EOF


# apt-get update
```

现在可以安装 Docker 引擎和各种 Kubernetes 的工具了。需要安装以下内容：

- Kubelet 可执行文件。
- Kubeadm 命令行界面。
- Kubectl 客户端。
- Kubernetes-cni：容器网络界面（Container Networking Interface,CNI）插件。

可以通过如下命令进行安装：

```
# apt-get install -y docker.io
# apt-get install -y kubelet kubeadm kubectl kubernetes-cni
```

讨论

在所有可执行文件和工具都安装完毕之后，就可以开始建立 Kubernetes 集群了。你可以通过下面的命令，在主节点上初始化集群：

```
# kubeadm init
[kubeadm] WARNING: kubeadm is in beta, please do not use it for production
clusters.
[init] Using Kubernetes version: v1.7.8
```

```
[init] Using Authorization modes: [Node RBAC]
[preflight] Running pre-flight checks
...
```

初始化的最后会给出一条命令，需要在所有工作节点上运行该命令（请参阅2.2节）。这个命令使用初始化进程自动生成的token。

请参阅

- 使用kubeadm创建集群（*https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/*）。

2.2 使用kubeadm创建Kubernetes集群

问题

在初始化Kubernetes的主节点之后（请参阅2.1节），如何向集群添加工作节点？

解决方案

按照2.1节所介绍的方法配置好Kubernetes的软件包仓库，并安装`kubeadm`，然后就可以在各个工作节点上运行如下`join`命令，并提供在主节点上运行`init`命令时生成的`token`：

```
$ kubeadm join --token <token>
```

下面返回主节点的终端，查看加进来的节点：

```
$ kubectl get nodes
```

讨论

最后一步需要创建满足 Kubernetes 要求的网络，尤其是需要给每个 pod 分配一个 IP 地址。你可以使用任何网络插件[注1]。例如，可以在 Kubernetes 集群 v1.6.0 或更高的版本上，通过下面的 kubectl 命令安装 Weave Net[注2]：

```
$ export kubever=$(kubectl version | base64 | tr -d '\n')
$ kubectl apply -f https://cloud.weave.works/k8s/net?k8s-version=$kubever
```

这个命令可以创建运行在集群所有节点之上的服务集（请参阅 7.3 节）。这些服务集使用主机的网络和一个 CNI（*https://github.com/containernetworking/cni*）插件来配置本地节点的网络。准备好网络后，集群节点就进入了 READY 状态。

在使用 kubeadm 启动集群的过程中，还可以利用其他插件来创建 pod 网络。请参阅下面的文档：

> *https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/#pod-network*

请参阅

- 使用 kubeadm 创建集群的相关文档（*https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/*）。

注 1：请参阅文档：Kubernetes，“Installing Addons”（*https://kubernetes.io/docs/concepts/cluster-administration/addons/*）。

注 2：请参阅详细文档：Weaveworks，“Integrating Kubernetes via the Addon”（*https://www.weave.works/docs/net/latest/kubernetes/kube-addon/*）。

2.3 从 GitHub 上下载 Kubernetes

问题

如何下载 Kubernetes 的官方版本，而不用从源代码编译？

解决方案

可以按照手动安装流程，前往 GitHub 提供的 Kubernetes 发布页面。选择想下载的发行版本（或预发行版本）。然后选择需要编译的源代码包，或下载 *kubernetes.tar.gz* 文件。

你也可以用 GitHub 的 API 检查最新版本，如下所示：

```
$ curl -s https://api.github.com/repos/kubernetes/kubernetes/releases | \
          jq -r .[].assets[].browser_download_url
https://github.com/kubernetes/kubernetes/releases/download/v1.9.0/
        kubernetes.tar.gz
https://github.com/kubernetes/kubernetes/releases/download/v1.9.0-beta.2/
        kubernetes.tar.gz
https://github.com/kubernetes/kubernetes/releases/download/v1.8.5/
        kubernetes.tar.gz
https://github.com/kubernetes/kubernetes/releases/download/v1.9.0-beta.1/
        kubernetes.tar.gz
https://github.com/kubernetes/kubernetes/releases/download/v1.7.11/
        kubernetes.tar.gz
...
```

然后下载所需的 *kubernetes.tar.gz* 发行包。例如，可以运行如下命令下载 v1.7.11：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/ \
       v1.7.11/kubernetes.tar.gz
```

如果你想从源代码编译 Kubernetes，请参阅 13.1 节。

请不要忘记验证 *kubernetes.tar.gz* 文件的 hash。GitHub 的发布页面给出了 SHA256 hash。在将压缩文件下载到本地后，请生成 hash 并进行比对。即便版本没有 GPG 签名，也可以通过验证 hash 检查压缩文件的完整性。

2.4 下载客户端和服务器端可执行文件

问题

下载的 Kubernetes 压缩文件（请参阅 2.3 节）中不包含实际的可执行文件，该怎么办？

解决方案

为了减小压缩文件的大小，可执行文件并没有被包含在内，所以需要单独下载。可以通过运行 *get-kube-binaries.sh* 脚本来下载可执行文件，命令如下：

```
$ tar -xvf kubernetes.tar.gz
$ cd kubernetes/cluster
$ ./get-kube-binaries.sh
```

运行上述命令可以将客户端的可执行文件下载到 *client/bin* 目录内：

```
$ tree ./client/bin
./client/bin
├── kubectl
└── kubefed
```

而服务器端的可执行文件则在 *server/kubernetes/server/bin* 目录内：

```
$ tree server/kubernetes/server/bin
server/kubernetes/server/bin
```

```
├── cloud-controller-manager
├── kube-apiserver
...
```

如果想跳过下载最新发行版本，直接下载客户端和服务器端的可执行文件，则可以直接从 *https://dl.k8s.io* 上下载。例如，通过下面的命令可以获取 v1.7.11 的 Linux 可执行文件：

```
$ wget https://dl.k8s.io/v1.7.11/ \
  \kubernetes-client-linux-amd64.tar.gz

$ wget https://dl.k8s.io/v1.7.11/ \
  kubernetes-server-linux-amd64.tar.gz
```

2.5 使用 hyperkube 映像通过 Docker 运行 Kubernetes 主节点

问题

如何使用一些 Docker 容器创建 Kubernetes 的主节点？具体来说，如何运行容器内的 API 服务器、调度器、控制器以及 etcd 键值数据库？

解决方案

使用 hyperkube 可执行文件，外加一个 etcd 容器。hyperkube 是以 Docker 映像方式提供的一站式可执行文件。你可以利用它启动所有的 Kubernetes 进程。

为了创建 Kubernetes 集群，你需要一个存储解决方案来保存集群的状态。Kubernetes 使用的解决方案是一个名叫 etcd 的键值分布式存储。因此，首先需要启动一个 etcd 的实例。可以运行如下命令：

```
$ docker run -d \
        --name=k8s \
        -p 8080:8080 \
        gcr.io/google_containers/etcd:3.1.10 \
        etcd --data-dir /var/lib/data
```

接下来可以通过一个名为 hyperkube 的映像文件启动 API 服务器，这个映像包含了 API 服务器的可执行文件，它保存在 Google Container Registry（简称 GCR）中，具体位置为 gcr.io/google_containers/hyperkube:v1.7.11。我们可以通过一些设置，在本地端口上提供非常安全的 API 服务。注意，v1.7.11 对应你机器上安装的 hyperkube 的版本：

```
$ docker run -d \
        --net=container:k8s \
        gcr.io/google_containers/hyperkube:v1.7.11/ \
        apiserver --etcd-servers=http://127.0.0.1:2379 \
        --service-cluster-ip-range=10.0.0.1/24 \
        --insecure-bind-address=0.0.0.0 \
        --insecure-port=8080 \
        --admission-control=AlwaysAdmit
```

最后，你可以启动指向 API 服务器的管理控制器：

```
$ docker run -d \
        --net=container:k8s \
        gcr.io/google_containers/hyperkube:v1.7.11/ \
        controller-manager --master=127.0.0.1:8080
```

请注意，由于 etcd、API 服务器和控制器管理共享同一个网络命名空间，因此尽管它们在不同的容器上运行，但是可以通过 127.0.0.1 互相访问。

为了测试设置的正确性，可以在 etcd 容器内使用 etcdctl，查看 */registry* 目录：

```
$ docker exec -ti k8s /bin/sh
# export ETCDCTL_API=3
# etcdctl get "/registry/api" --prefix=true
```

你还可以通过下面的命令访问 Kubernetes API 服务器，并查看 API：

```
$ curl -s curl http://127.0.0.1:8080/api/v1 | more
{
  "kind": "APIResourceList",
  "groupVersion": "v1",
  "resources": [
    {
      "name": "bindings",
      "singularName": "",
      "namespaced": true,
      "kind": "Binding",
      "verbs": [
        "create"
      ]
    },
...
```

迄今为止，我们还没有介绍调度器，也未曾使用 kubelet 和 kube-proxy 建立节点。本节我们仅演示了如何通过启动本地的 3 个容器运行 Kubernetes API。

有时使用 hyperkube 的 Docker 映像来验证 Kubernetes 的可执行文件的配置选项非常有用。例如，可以运行如下命令检查 /apiserver 命令的帮助文档：

```
$ docker run --rm -ti \
        gcr.io/google_containers/hyperkube:v1.7.11 \
        /apiserver --help
```

讨论

尽管在本地探索 Kubernetes 的组件非常有帮助，但是在搭建生产环境时我们不推荐这样做。

请参阅

- hyperkube 的 Docker 映像（*https://github.com/kubernetes/kubernetes/tree/master/cluster/images/hyperkube*）。

2.6 编写 systemd 单元文件来运行 Kubernetes 的组件

问题

你已经学习了如何使用 Minikube（请参阅 1.3 节），并且掌握了如何使用 kubeadm 建立 Kubernetes 集群（请参阅 2.2 节），但还需要了解如何从头安装一个集群。要做到这一点，你需要使用 systemd 单元文件来运行 Kubernetes 的组件。我们通过一个基本的例子来说明如何利用 systemd 运行 kubelet。

解决方案

Systemd[注3] 系统是服务管理器，有时也被称为初始化系统。Ubuntu 16.04 和 CentOS 7 将它作为默认的服务管理器。

你可以参照 kubeadm 的方法来自己动手建立集群。如果仔细观察 kubeadm 的配置，就会发现集群中每个节点上运行的 kubelet，包括主节点上的 kubelet，都是由 systemd 管理的。

在下面的例子中，登录由 kubeadm 创建的集群（请参阅 2.2 节）中的任意节点，执行如下命令，都可以看到类似的信息：

```
# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset:
           enabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Tue 2017-06-13 08:29:33 UTC; 2 days ago
     Docs: http://kubernetes.io/docs/
 Main PID: 4205 (kubelet)
```

注 3：请参阅文档：freedesktop.org“systemd”（*https://www.freedesktop.org/wiki/Software/systemd/*）。

```
    Tasks: 17
   Memory: 47.9M
      CPU: 2h 2min 47.666s
   CGroup: /system.slice/kubelet.service
           ├─ 4205 /usr/bin/kubelet --kubeconfig=/etc/kubernetes/kubelet.conf \
           |                        --require-kubeconfig=true \
           |                        --pod-manifest-path=/etc/kubernetes/manifests \
           |                        --allow-privileged=true \
           |                        --network-plugin=cni \
           |                        --cni-conf
           └── 4247 journalctl -k -f
```

从以上命令返回的结果中可以看到 systemd 的单元文件位于 */lib/systemd/system/kubelet.service* 目录之中，而配置文件位于 */etc/systemd/system/kubelet.service.d/10-kubeadm.conf* 目录。

单元文件简单明了，它指向安装在 */usr/bin* 下的 kubelet 可执行文件：

```
[Unit]
Description=kubelet: The Kubernetes Node Agent
Documentation=http://kubernetes.io/docs/

[Service]

ExecStart=/usr/bin/kubelet
Restart=always
StartLimitInterval=0
RestartSec=10

[Install]
WantedBy=multi-user.target
```

配置文件展示了如何启动 kubelet 可执行文件：

```
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--kubeconfig=/etc/kubernetes/kubelet.conf
             --require-kubeconfig=true"
Environment="KUBELET_SYSTEM_PODS_ARGS=--pod-manifest-path=/etc/kubernetes/
             manifests --allow-privileged=true"
Environment="KUBELET_NETWORK_ARGS=--network-plugin=cni
             --cni-conf-dir=/etc/cni/net.d --cni-bin-dir=/opt/cni/bin"
Environment="KUBELET_DNS_ARGS=--cluster-dns=10.96.0.10
             --cluster-domain=cluster.local"
Environment="KUBELET_AUTHZ_ARGS=--authorization-mode=Webhook
             --client-ca-file=/etc/kubernetes/pki/ca.crt"
ExecStart=
```


```
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_SYSTEM_PODS_ARGS
          $KUBELET_NETWORK_ARGS $KUBELET_DNS_ARGS $KUBELET_AUTHZ_ARGS
          $KUBELET_EXTRA_ARGS
```

上述所有参数，例如通过环境变量$KUBELET_CONFIG_ARGS指定的--kubeconfig等，都是启动kubelet的参数。

讨论

这个单元文件只负责kubelet。你可以为所有其他Kubernetes集群的组件编写单元文件，例如API服务、控制器管理、调度器、代理等。Kubernetes the Hard Way提供了每个组件的单元文件的例子[注4]。

然而，你只需要运行kubelet。事实上，你可以通过配置参数--podmanifest-path把自动启动清单文件（manifests）所在的路径传递给kubelet。在kubeadm内，这个路径用于传递API服务器、调度器和控制器管理的清单文件。因此Kubernetes可以自我管理，而唯一由systemd管理的是kubelet进程。

为了证实这一点，你可以通过如下命令查看kubeadm创建的集群*/etc/kubernetes/manifests*目录中的内容：

```
# ls -l /etc/kubernetes/manifests
total 16

-rw------- 1 root root 1071 Jun 13 08:29 etcd.yaml
-rw------- 1 root root 2086 Jun 13 08:29 kube-apiserver.yaml
-rw------- 1 root root 1437 Jun 13 08:29 kube-controller-manager.yaml
-rw------- 1 root root  857 Jun 13 08:29 kube-scheduler.yaml
```

仔细观察*etcd.yaml*清单文件，就会发现这个Pod只有一个容器，里面运行etcd：

注4：请参阅文档：Kubernetes the Hard Way，“Bootstrapping the Kubernetes Control Plane（*https://github.com/kelseyhightower/kubernetes-the-hard-way/blob/master/docs/08-bootstrapping-kubernetes-controllers.md?*）。

```
# cat /etc/kubernetes/manifests/etcd.yaml

apiVersion:          v1
kind:                Pod
metadata:
  creationTimestamp: null
  labels:
    component:       etcd
    tier:            control-plane
  name:              etcd
  namespace:         kube-system
spec:
  containers:
  - command:
    - etcd
    - --listen-client-urls=http://127.0.0.1:2379
    - --advertise-client-urls=http://127.0.0.1:2379
    - --data-dir=/var/lib/etcd
    image:           gcr.io/google_containers/etcd-amd64:3.0.17
...
```

请参阅

- Kubelet 的配置参数（*https://kubernetes.io/docs/reference/generated/kubelet/*）。

2.7 在 Google Kubernetes 引擎上创建 Kubernetes 集群

问题

如何在 Google Kubernetes 引擎（Google Kubernetes Engine，简称 GKE）上创建 Kubernetes 集群？

解决方案

可以使用 **gcloud** 命令行界面，通过 `container clusters create` 命令创建 Kubernetes 的集群，如下所示：

```
$ gcloud container clusters create oreilly
```

默认情况下，上述命令创建的集群自带 3 个工作节点。主节点由 GKE 服务管理，无法访问。

讨论

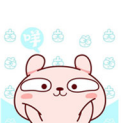
为了使用 GKE，首先需要完成以下几项：

- 在 Google 云平台创建并激活账户。
- 创建一个项目，并在里面激活 GKE 服务。
- 在自己的机器上安装 gcloud 命令行界面。

你可以利用 Google Cloud Shell 快速设置 `gcloud`，Google Cloud Shell 是完全基于浏览器的在线解决方案，相关文档链接（*https://cloud.google.com/shell/docs/*）。

在集群创建好后，可以通过如下命令查看集群：

```
$ gcloud container clusters list
NAME     ZONE            MASTER_VERSION  MASTER_IP      ...  STATUS
oreilly  europe-west1-b  1.7.8-gke.0     35.187.80.94   ...  RUNNING
```



gcloud 允许你设置集群的大小，以及更新和升级集群：

```
...
COMMANDS
...
     resize
        Resizes an existing cluster for running containers.
     update
        Update cluster settings for an existing container cluster.
     upgrade
        Upgrade the Kubernetes version of an existing container
        cluster.
```

在使用完集群后，请别忘了删除，避免产生费用：

```
$ gcloud container clusters delete oreilly
```

请参阅

- GKE 快速入门（*https://cloud.google.com/kubernetes-engine/docs/quickstart*）。
- Google Cloud Shell 快速入门（*https://cloud.google.com/shell/docs/quickstart*）。

2.8 在 Azure 容器服务上创建 Kubernetes 集群

问题

如何在 Azure 容器服务（Azure Container Service，ACS）上创建 Kubernetes 集群？

解决方案

为了执行以下步骤，首先需要注册一个（免费的）Azure 账号（*https://azure.*

microsoft.com/en-us/free/），并安装 Azure 命令行版本 2.0（*https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest*）。

首先，请确认你安装了正确的命令行版本，然后登录：

```
$ az --version | grep ^azure-cli
azure-cli (2.0.13)

$ az login
To sign in, use a web browser to open the page https://aka.ms/devicelogin and
enter the code XXXXXXXXX to authenticate.
[
  {
    "cloudName": "AzureCloud",
    "id": "****************************",
    "isDefault": true,
    "name": "Free Trial",
    "state": "Enabled",
    "tenantId": "****************************",
    "user": {
      "name": "******@hotmail.com",
      "type": "user"
    }
  }
]
```

可以预先准备一个叫做 k8s 的 Azure 资源组（相当于 Google Could 的项目）。这个资源组负责保存所有的资源，包括虚拟机和网络组件，并方便日后清理或删除：

```
$ az group create --name k8s --location northeurope
{
  "id": "/subscriptions/************************/resourceGroups/k8s",
  "location": "northeurope",
  "managedBy": null,
  "name": "k8s",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null
}
```


如果你不清楚 `--location` 参数应该设置什么区域，请运行 `az account list-locations` 命令，并选择离你最近的区域。

设置好 k8s 资源组之后，可以在一个工作节点上创建集群（Azure 称之为代理 *agent*），具体命令如下：

```
$ az acs create --orchestrator-type kubernetes \
                --resource-group k8s \
                --name k8scb \
                --agent-count 1 \
                --generate-ssh-keys
waiting for AAD role to propagate.done
{
...
"provisioningState": "Succeeded",
"template": null,
"templateLink": null,
"timestamp": "2017-08-13T19:02:58.149409+00:00"
},
"resourceGroup": "k8s"
}
```

请注意 `az acs create` 命令可能需要 10 分钟才能完成。

如果你的 Azure 免费账号没有足够的配额创建一个默认的（3 个代理）Kubernetes 集群的话，那么你会见到如下错误提示信息：

```
Operation results in exceeding quota limits of Core. Maximum allowed:
4, Current in use: 0, Additional requested: 8.
```

为了解决这个问题，你可以选择创建一个小点的集群，例如使用参数 `--agent-count 1`，或使用收费订阅。

集群创建完毕后，可以在 Azure 门户网站看到如图 2-1 所示的信息。首先可以先找到 k8s 资源群，然后通过 Deployments 标签找到集群。

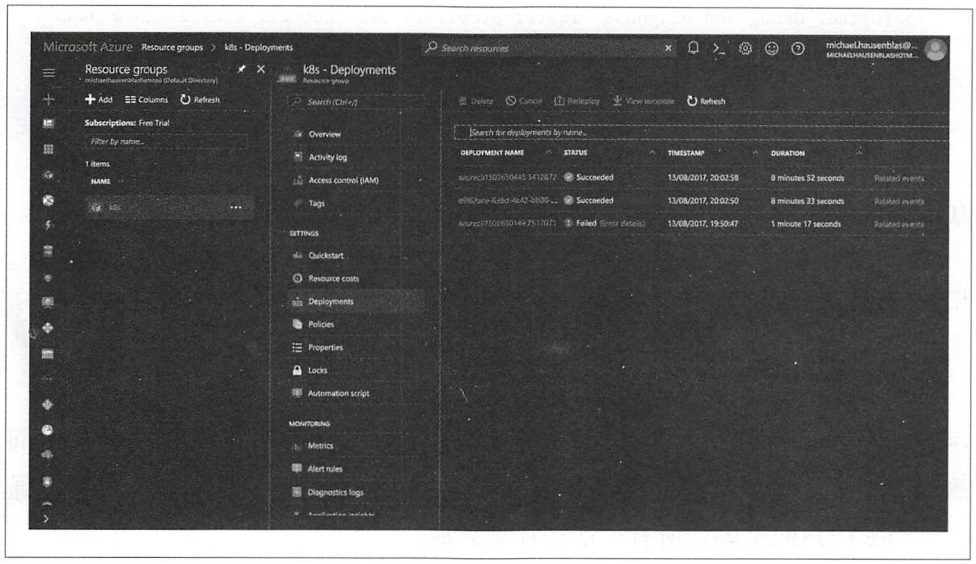


图 2-1：Azure 门户网站的截图，显示了 k8s 资源组中 ACS 的部署信息

现在可以通过下面的命令连接集群了：

```
$ az acs kubernetes get-credentials --resource-group=k8s --name=k8scb
```

你可以通过下面的命令查看环境，确认设置：

```
$ kubectl cluster-info
Kubernetes master is running at https://k8scb-k8s-143f1emgmt.northeurope.cloudapp
  .azure.com
Heapster is running at https://k8scb-k8s-143f1emgmt.northeurope.cloudapp.azure
  .com/api/v1/namespaces/kube-system/services/heapster/proxy
KubeDNS is running at https://k8scb-k8s-143f1emgmt.northeurope.cloudapp.azure
  .com/api/v1/namespaces/kube-system/services/kube-dns/proxy
kubernetes-dashboard is running at https://k8scb-k8s-143f1emgmt.northeurope
  .cloudapp.azure.com/api/v1/namespaces/kube-system/services/kubernetes-dashboard
  /proxy
tiller-deploy is running at https://k8scb-k8s-143f1emgmt.northeurope.cloudapp
  .azure.com/api/v1/namespaces/kube-system/services/tiller-deploy/proxy
```

```
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

$ kubectl get nodes
NAME                    STATUS                     AGE       VERSION
k8s-agent-1a7972f2-0    Ready                      7m        v1.7.8
k8s-master-1a7972f2-0   Ready,SchedulingDisabled   7m        v1.7.8
```

从上面的输出结果可以看到，确实有一个代理（工作）节点和一个主节点。

用完 ACS 之后，别忘了关闭集群，并通过删除资源组 k8s 移除所有的资源：

```
$ az group delete --name k8s --yes --no-wait
```

尽管 az group delete 命令会立即返回，但是事实上需要花费 10 分钟的时间删除虚拟机、虚拟网络或硬盘等所有资源，以及真正摧毁资源群。你可以通过 Azure 门户网站确认所有一切按计划完成。

如果你不想或无法安装 Azure 命令行界面，那么可以在浏览器中使用 Azure Cloud Shell（*https://azure.microsoft.com/en-us/features/cloud-shell/*），这样就不用执行安装 Kubernetes 集群的步骤了。

请参阅

- 微软 Azure 文档：为 Linux 容器部署 Kubernetes 集群（*https://docs.microsoft.com/en-us/azure/container-service/kubernetes/container-service-kubernetes-walkthrough*）。

第 3 章

学习使用 Kubernetes 客户端

本章的各小节将介绍 Kubernetes 命令行界面，以及 kubectl 的基本用法。关于如何安装命令行界面工具请参照第 1 章；有关高级用法，例如 Kubernetes API 的用法请参照第 6 章。

3.1 查看资源

问题

如何查看 Kubernetes 的某种资源？

解决方案

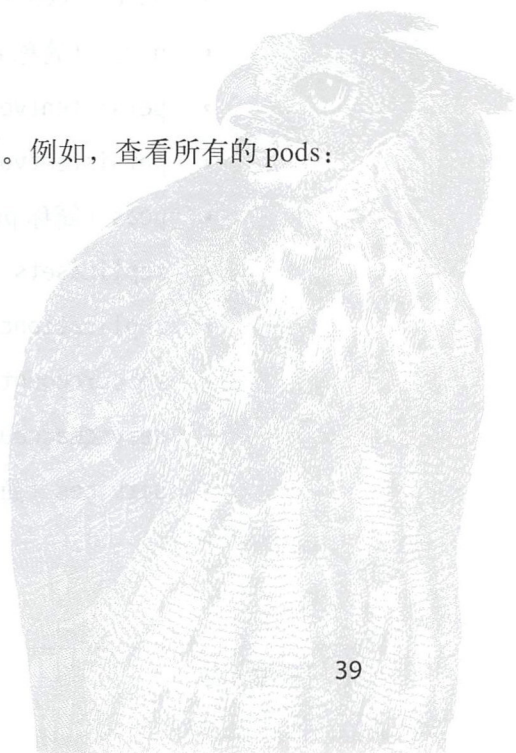
可以使用 kubectl 的 get 方法，并指明资源类型。例如，查看所有的 pods：

```
$ kubectl get pods
```

查看所有的服务和部署：

```
$ kubectl get services,deployments
```

查看某个特定的部署：



```
$ kubectl get deployment myfirstk8sapp
```

查看所有的资源：

```
$ kubectl get all
```

kubectl 的 get 方法是最基本但非常实用的命令，可以用于快速查看集群的状况，基本上相当于 UNIX 的 ps。

很多资源都有简称，你可以在 kubectl 命令中使用这些简称，以节约时间，也可以减少出错。下面是一些例子：

- configmaps（简称 cm）。
- daemonsets（简称 ds）。
- deployments（简称 deploy）。
- endpoints（简称 ep）。
- events（简称 ev）。
- horizontalpodautoscalers（简称 hpa）。
- ingresses（简称 ing）。
- namespaces（简称 ns）。
- nodes（简称 no）。
- persistentvolumeclaims（简称 pvc）。
- persistentvolumes（简称 pv）。
- pods（简称 po）。
- replicasets（简称 rs）。
- replicationcontrollers（简称 rc）。
- resourcequotas（简称 quota）。
- serviceaccounts（简称 sa）。
- services（简称 svc）。

3.2 删除资源

问题

如何删除不需要的资源？

解决方案

可以使用kubectl的delete方法，并指明资源类型，以及希望删除的资源名称。

例如，删除命名空间my-apps中的所有资源：

```
$ kubectl get ns
NAME          STATUS    AGE
default       Active    2d
kube-public   Active    2d
kube-system   Active    2d

my-app        Active    20m

$ kubectl delete ns my-app
namespace "my-app" deleted
```

关于如何创建一个命名空间，请参阅6.3节。

你还可以删除特定的资源，同时影响与资源相关的进程。例如，删除标签为app=niceone的服务以及部署的命令如下所示：

```
$ kubectl delete svc,deploy -l app=niceone
```

下面的命令强行删除了一个pod：

```
$ kubectl delete pod hangingpod --grace-period=0 --force
```

删除test命名空间中的所有pod：

```
$ kubectl delete pods --all --namespace test
```

讨论

不要直接删除被监控的对象，例如由部署控制的 pod 等。应该先关闭它们的监控进程，或使用特定的操作删掉被管理的资源。例如，可以先将一个部署缩小到零个副本（请参阅 9.1 节），然后就可以有效地删除它监控的所有 pod 了。

另外需要考虑的一点是级联删除与直接删除。例如，当删除一个自定义的资源定义（CRD，请参阅 13.4 节）时，其所有的依赖对象也会被删除。更多关于级联删除规则的信息，请参照 Kubernetes 的文档（*https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/*）。

3.3 使用 kubectl 观察资源的变化

问题

如何通过终端以交互模式观察 Kubernetes 对象的变化？

解决方案

Kubectl 命令有一个 `--watch` 的选项可以帮助你做到这一点。例如，观察 pod 的命令如下：

```
$ kubectl get pods --watch
```

请注意这个命令将独占前台，并实时显示资源的占用状况，类似于 `top` 命令。

讨论

`--watch` 选项非常实用，但在刷新屏幕的时候，有时不太可靠。另一种方法是使用 `watch` 命令，如下所示：

```
$ watch kubectl get pods
```

3.4 使用 kubectl 编辑资源

问题

如何更新 Kubernetes 资源的属性？

解决方案

可以使用 `kubectl` 的 `edit` 方法，并指明资源类型：

```
$ kubectl run nginx --image=nginx
$ kubectl edit deploy/nginx
```

然后在编辑器内编辑 nginx，比如将副本数变更为 2。保存之后，系统会提示：

```
deployment "nginx" edited
```

讨论

如果遇到编辑器问题，可以设置 `EDITOR=vi`。请注意并不是所有的变更都会触发一次部署。

部分触发器有快捷键，例如，如果想改变某个部署使用的映像版本，那么可以简单地使用 kubectl set image，这个命令可以更新已有容器的映像的资源（对部署、副本集或副本控制器、服务进程集、任务以及简单的 pod 都有效）。

3.5 通过 kubectl 解释资源和字段

问题

如何深入了解某个资源（比如 service），或了解 Kubernetes 内某个字段的含义及其默认值，以及该字段是必须的还是可选的？

解决方案

可以使用 kubectl 的 explain 命令，如下所示：

```
$ kubectl explain svc
DESCRIPTION:
Service is a named abstraction of software service (for example, mysql)
consisting of local port (for example 3306) that the proxy listens on, and the
selector that determines which pods will answer requests sent through the proxy.

FIELDS:
   status       <Object>
     Most recently observed status of the service. Populated by the system.
     Read-only. More info: https://git.k8s.io/community/contributors/devel/
     api-conventions.md#spec-and-status/

   apiVersion   <string>
     APIVersion defines the versioned schema of this representation of an
     object. Servers should convert recognized schemas to the latest internal
     value, and may reject unrecognized values. More info:
     https://git.k8s.io/community/contributors/devel/api-conventions.md#resources

   kind <string>
     Kind is a string value representing the REST resource this object
     represents. Servers may infer this from the endpoint the client submits
     requests to. Cannot be updated. In CamelCase. More info:
     https://git.k8s.io/community/contributors/devel/api-conventions
     .md#types-kinds

   metadata     <Object>
     Standard object's metadata. More info:
     https://git.k8s.io/community/contributors/devel/api-conventions.md#metadata
```

```
   spec <Object>
     Spec defines the behavior of a service. https://git.k8s.io/community/
     contributors/devel/api-conventions.md#spec-and-status/

$ kubectl explain svc.spec.externalIP
FIELD: externalIPs <[]string>

DESCRIPTION:
     externalIPs is a list of IP addresses for which nodes in the cluster will
     also accept traffic for this service.  These IPs are not managed by
     Kubernetes.  The user is responsible for ensuring that traffic arrives at a
     node with this IP.  A common example is external load-balancers that are not
     part of the Kubernetes system.
```

讨论

`kubectl` 的 `explain` 命令[注1]从 Swagger/OpenAPI 定义[注2]中摘取由 API 服务器提供的资源和字段的描述。

请参阅

- Ross Kukulinski 的博文（*https://blog.heptio.com/kubectl-explain-heptioprotip-ee883992a243*）。

注 1：请参阅文档：Kubernetes，“Kubectl Reference Docs: Explain”（*https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#explain*）。

注 2：请参阅文档：Kubernetes，“The Kubernetes API”（*https://kubernetes.io/docs/concepts/overview/kubernetes-api/*）。

第 4 章

创建与修改基础的工作负载

本章中，我们将介绍如何管理 Kubernetes 基础的工作负载类型：pod 与部署。我们将介绍如何通过命令行界面的命令和 YAML 的清单文件创建部署和 pod，还将介绍如何伸缩部署和更新部署。

4.1 通过 kubectl run 创建部署

问题

如何快速启动一个长期运行的应用程序，例如 Web 服务器？

解决方案

可以使用 kubectl run 命令，这个命令是即刻创建部署的清单文件的生成器。例如，创建一个部署来运行 Ghost 微型博客平台，其具体命令如下：

```
$ kubectl run ghost --image=ghost:0.9

$ kubectl get deploy/ghost
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
ghost     1         1         1            0           16s
```

讨论

kubectl run 命令拥有许多选项可以用于设置部署的参数。例如，你可以设置如下参数：

- 通过 --env 设置环境变量。
- 通过 --port 定义容器的端口。
- 通过 --command 定义运行的命令。
- 通过 --expose 自动创建一个关联的服务。
- 通过 --replicas 定义 pod 的数量。

常见的使用方法如下所示。例如，在端口 2368 上启动 Ghost，并随之创建一个服务：

```
$ kubectl run ghost --image=ghost:0.9 --port=2368 --expose
```

用 root 的密码启动 MySQL：

```
$ kubectl run mysql --image=mysql:5.5 --env=MYSQL_ROOT_PASSWORD=root
```

启动一个运行 busybox 的容器，并在启动的时候执行 sleep 3600：

```
$ kubectl run myshell --image=busybox --command -- sh -c "sleep 3600"
```

通过运行 kubectl run --help 命令可以查看关于各个参数的详细信息。

4.2 通过清单文件创建对象

问题

如何在不借助 kubectl run 等生成器的情况下，通过明确指定属性的方式创建对象？

解决方案

可以使用 kubectl create 命令完成这个操作：

```
$ kubectl create -f <manifest>
```

6.3 节中将介绍如何使用 YAML 的清单文件创建命名空间。这里介绍一个非常简单的清单文件的例子。清单文件可以用 YAML 或 JSON 书写，如下是一个用 YAML 编写的清单文件 *myns.yaml*：

```
apiVersion:    v1
kind:          namespace
metadata:
  name:        myns
```

接下来，可以通过 kubectl create -f myns.yaml 命令创建该对象。

讨论

可以将 kubectl create 命令指向某个 URL，或本地文件系统中的某个文件名。例如，为了创建典型的留言板应用程序的前端，你可以获取定义了该应用程序的唯一的 YAML 文件的 URL，并应用到创建命令中，如下所示：

```
$ kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes/ \
                    master/examples/guestbook/frontend-deployment.yaml
```

4.3 从零创建 pod 的清单文件

问题

如何在不借助 kubectl run 等生成器的情况下，从零编写 pod 的清单文件？

解决方案

Pod 是一个 `/api/v1` 的对象，与其他 Kubernetes 的对象类似，它的清单文件包含下列字段：

- `apiVersion`：指定 API 的版本。
- `kind`：指定对象类型。
- `metadata`：提供对象相关的元数据。
- `Spec`：提供对象的规格。

Pod 的清单文件包含一组容器，以及一组可选的卷（volume）（请参阅第 8 章）。这里介绍一个最简单的例子，只包含一个容器，不包含卷，如下所示：

```
apiVersion:    v1
kind:          Pod
metadata:
  name:        oreilly
spec:
  containers:
  - name:      oreilly
    image:     nginx
```

将这个 YAML 的清单文件保存到名为 *oreilly.yaml* 的文件中，然后利用 `kubectl` 创建 pod：

```
$ kubectl create -f oreilly.yaml
```

讨论

与上述解决方案中最基本的示例相比，真正的 pod 的 API 规格要复杂得多。例如，pod 可能包含多个容器，如下所示：

```
apiVersion: v1
kind:       Pod
metadata:
  name:     oreilly
spec:
  containers:
  - name:   oreilly
    image:  nginx
  - name:   safari
    image:  redis
```

pod 还可能包含卷的定义，以便在容器中加载数据（请参阅 8.1 节），也可以探测容器化应用程序的健康状况（请参阅 11.2 节与 11.3 节）。

更多关于规格字段的描述，以及 API 对象的所有规格，请参阅如下文档（*https://kubernetes.io/docs/concepts/workloads/pods/pod/*）。

除非有特殊原因，否则不要单独创建 pod。请使用一个部署对象（请参阅 4.4 节）监管 pod，它会通过另外一个叫做 ReplicaSet（副本集）的对象监控 pod。

请参阅

- Kubernetes Pod 的参考文档（*https://v1-7.docs.kubernetes.io/docs/reference/#pod-v1-core*）。
- ReplicaSet 文档（*https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/*）。

4.4 通过 kubectl run 创建部署

问题

如何全面控制（长期运行的）应用的启动与监控？

解决方案

编写一个清单文件定义 Deployment 对象。基本方法请参阅 4.3 节。

假设我们创建一个叫 *fancyapp.yaml* 的清单文件，内容如下：

```
apiVersion:       extensions/v1beta1
kind:             Deployment
metadata:
  name:           fancyapp
spec:
  replicas:       5
  template:
    metadata:
      labels:
        app:      fancy
        env:      development
    spec:
      containers:
      - name:     sise
        image:    mhausenblas/simpleservice:0.5.0
        ports:
        - containerPort: 9876
        env:
        - name:   SIMPLE_SERVICE_VERSION
          value:  "0.9"
```

可以看出，启动该应用的时候，有几件事情需要明确指定：

- 设定需要启动与监控的 pod（replica，即完全相同的副本）的数目。
- 贴上标签，比如 env=development（请参阅 6.5 节与 6.6 节）。
- 设置环境变量，比如 SIMPLE_SERVICE_VERSION。

现在让我们来看看部署包含的内容：

```
$ kubectl create -f fancyapp.yaml
deployment "fancyapp" created

$ kubectl get deploy
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
fancyapp   5         5         5            0           8s
```

```
$ kubectl get rs
NAME                  DESIRED   CURRENT   READY     AGE
fancyapp-1223770997   5         5         0         13s

$ kubectl get po
NAME                        READY     STATUS              RESTARTS   AGE
fancyapp-1223770997-18msl   0/1       ContainerCreating   0          15s
fancyapp-1223770997-1zdg4   0/1       ContainerCreating   0          15s
fancyapp-1223770997-6rqn2   0/1       ContainerCreating   0          15s
fancyapp-1223770997-7bnbh   0/1       ContainerCreating   0          15s
fancyapp-1223770997-qxg4v   0/1       ContainerCreating   0          15s
```

几秒钟后再查看一次：

```
$ kubectl get po
NAME                        READY     STATUS    RESTARTS   AGE
fancyapp-1223770997-18msl   1/1       Running   0          1m
fancyapp-1223770997-1zdg4   1/1       Running   0          1m
fancyapp-1223770997-6rqn2   1/1       Running   0          1m
fancyapp-1223770997-7bnbh   1/1       Running   0          1m
fancyapp-1223770997-qxg4v   1/1       Running   0          1m
```

如果想要删除一个部署及其副本集与监管的 pod，那么可以执行 `kubectl delete deploy/fancyapp` 命令。请不要单独删除 pod，因为部署会重建 pod。初学者常常会因此而感到困惑。

部署允许用户伸缩应用的规模（请参阅 9.1 节），也可以升级到新版本或回滚到前一个版本。一般来说，对于需要多个具有相同特征的 pod 的无状态应用来说，这些方法非常实用。

讨论

部署负责监管 pod 和副本集（Replica Sets，RS）可以允许你细致地控制如何以及何时应该将 pod 升级到新版本，或回滚到之前的版本。一般来说，你无需在意部署监管的副本集与 pod，除非某些特殊情况，比如需要调试某个 pod（请参阅 12.5 节）。图 4-1 展示了如何在部署不同的版本间来回切换。

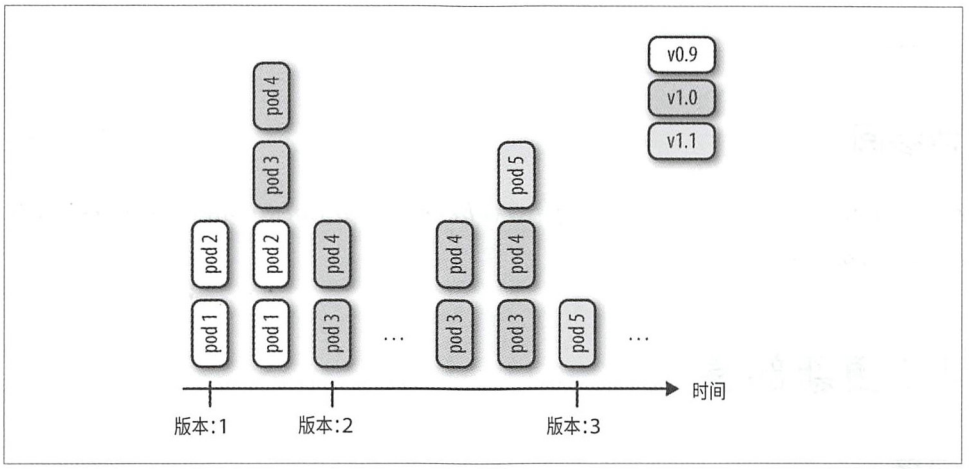


图 4-1：部署的版本

请注意，以后副本集将取代原来的副本控制器 (Replication Controller，RC)，所以最好现在就开始考虑使用副本集。虽然目前两者唯一的区别在于副本集支持以集合为基础的标签与查询，但是以后会有更多功能加入到副本集，而副本控制器将逐步退出舞台。

最后，你可以使用 `kubectl create` 命令以及 `--dry-run` 选项生成清单文件。首先可以生成 YAML 或 JSON 格式的清单文件，然后保存下来方便以后使用。例如，下面的命令使用 `nginx` 的 Docker 映像创建名为 `fancy-app` 的部署的清单文件：

```
$ kubectl create deployment fancyapp --image nginx -o json --dry-run
{
    "kind": "Deployment",
    "apiVersion": "extensions/v1beta1",
    "metadata": {
        "name": "fancy-app",
        "creationTimestamp": null,
        "labels": {
            "app": "fancy-app"
        }
```

```
    },
...
```

请参阅

- Kubernetes 部署的介绍（*https://kubernetes.io/docs/concepts/workloads/controllers/deployment/*）。

4.5 更新部署

问题

如何将部署更新到新版本的应用程序？

解决方案

只需更新部署，并使用默认的更新策略 RollingUpdate 自动处理更新。

例如，想要更新一个容器映像的部署，可以运行如下命令：

```
$ kubectl run sise --image=mhausenblas/simpleservice:0.4.0
deployment "sise" created

$ kubectl set image deployment sise mhausenblas/simpleservice:0.5.0
deployment "sise" image updated


$ kubectl rollout status deployment sise
deployment "sise" successfully rolled out

$ kubectl rollout history deployment sise
deployments "sise"
REVISION        CHANGE-CAUSE
1               <none>
2               <none>
```

现在已经成功地推出了部署的新版本，实际上只更新了容器的映像。该部署的其他属性保持不变，如副本个数等。但是如果想要更新部署的其他方面，比如更新环境变量呢？那么可以使用多个 kubectl 命令更新该部署。例如，向当前部署添加一个端口定义，则可以使用 kubectl edit 命令：

```
$ kubectl edit deploy sise
```

这个命令将在默认的编辑器中打开当前部署，如果设置并导出了环境变量 KUBE_EDITOR 的话，将会在指定的编辑器中打开当前部署。

例如，添加下列端口定义：

```
...
  ports:
  - containerPort: 9876
...
```

编辑过程如图 4-2 所示，图中 KUBE_EDITOR 指定为 vi。

在保存并退出编辑后，Kubernetes 会启用一个新的部署，该部署将使用刚才定义的端口。我们可以确认一下：

```
$ kubectl rollout history deployment sise
deployments "sise"
REVISION        CHANGE-CAUSE
1               <none>
2               <none>
3               <none>
```

我们确实可以看到新推出的版本 3，其中包含了我们用 kubectl edit 添加的改动。CHANGE-CAUSE（变更原因）一栏之所以为空是因为没有使用 kubectl create 的 --record 选项。如果想了解什么变更触发了新的版本，可以加上此选项。

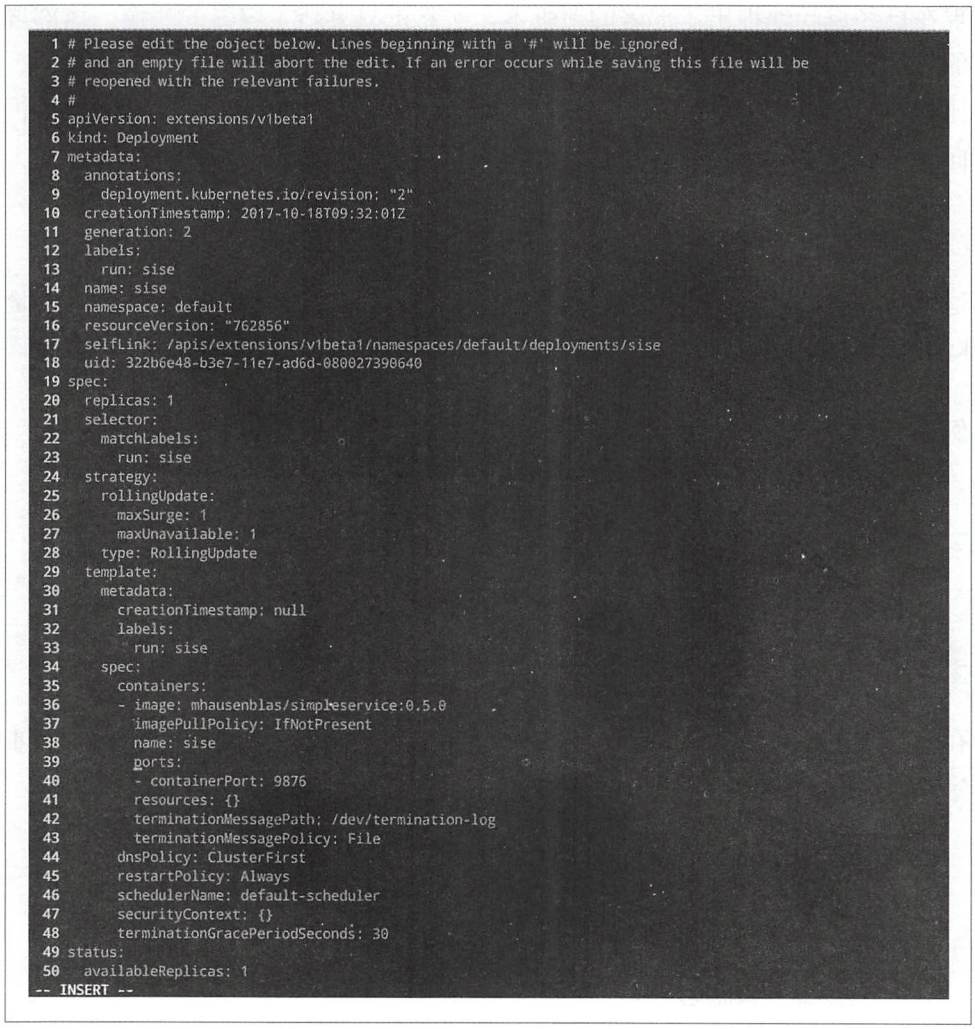


图 4-2：编辑部署

如前所述，还有很多 kubectl 命令可以用于更新部署：

- 使用 kubectl apply，根据清单文件更新（如果不存在可以创建）部署，比如 kubectl apply -f simpleservice.yaml。
- 使用 kubectl replace，根据清单文件替换部署，比如，kubectl replace -f

simpleservice.yaml。请注意 replace 的对象部署必须存在，这一点与 apply 命令不同。

- 使用 kubectl patch 更新特定的键值，比如：

```
kubectl patch deployment sise -p '{"spec": {"template":
{"spec": {"containers":
[{"name": "sise", "image": "mhausenblas/simpleservice:0.5.0"}]}}}}'
```

如果在新版本的部署中出错或遇到问题，那么该怎么办呢？很幸运的是，Kubernetes 的 kubectl rollout undo 命令可以很方便地回滚到已知的良好状态。比如，最后一次编辑出了错，想要回滚到版本 2，那么可以执行如下命令：

```
$ kubectl rollout undo deployment sise ??to?revision=2
```

可以通过 kubectl get deploy/sise -o yaml 确认端口定义已被删除。

只有 pod 模板（即 .spec.template 之下的键值）的变更才可以触发新的部署，例如：环境变量、端口或容器的映像。部署方面的变更，比如副本个数等，不会触发新的部署。

第 5 章

使用服务

本章中，我们将讨论集群内的 pod 如何互相通信，应用程序如何互相发现对方，以及如何公开 pod，以便从集群外访问。

我们这里用到的原语叫做 Kubernetes 服务（Kubernetes service），具体内容如图 5-1 所示。

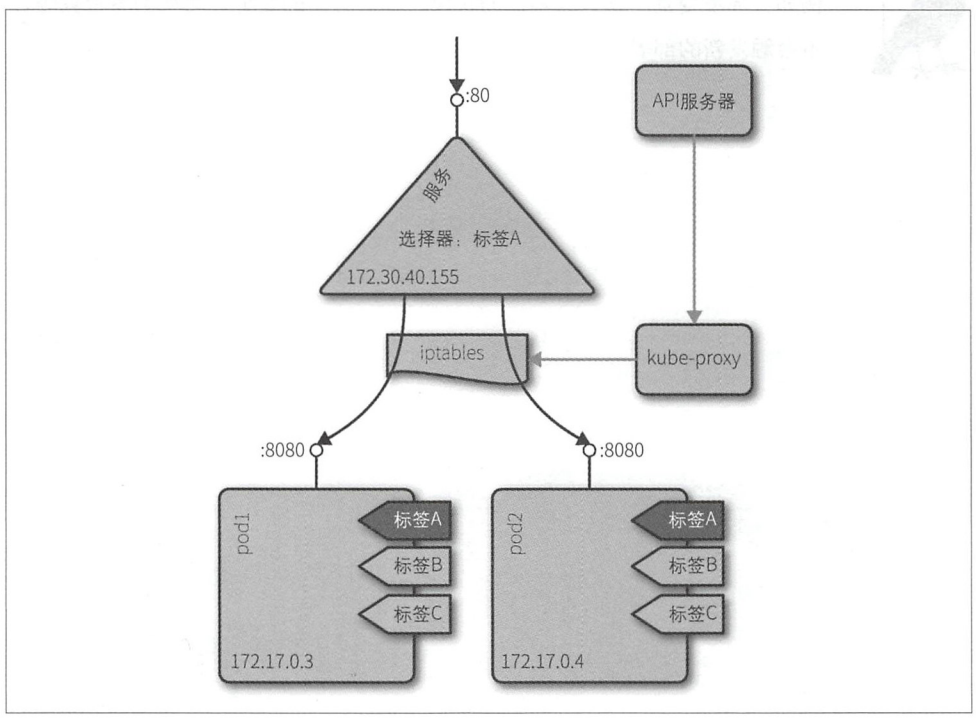


图 5-1：Kubernetes 服务的概念

服务可以为一组 pod 提供稳定的虚拟 IP 地址（Virtual IP，VIP）。即使加入新的 pod 或移除已有 pod，服务也能保证客户端可以通过 VIP 可靠地发现并链接到 pod 中运行的容器。VIP 中的“Virtual”的意思是说该 IP 并不是连接到网络接口的真实 IP 地址，它的目的只是为了将访问发送到一个或多个 pod。kube-proxy 负责维护 VIP 与 pod 之间的映射工作，集群的每个节点都需要运行该进程。kube-proxy 进程查询 API 服务器以获知集群中的新服务，并相应地更新节点的 iptables 规则，以提供必要的路由信息。

5.1 通过创建服务来公布应用程序

问题

如何在集群内提供一个稳定可靠的方法，以发现并访问应用程序？

解决方案

为构成应用程序的 pod 创建 Kubernetes 服务。

假设我们使用 kubectl run nginx --image nginx 创建了一个 nginx 部署，那么运行 kubectl expose 命令的时候，可以自动创建一个 Service 对象，如下所示：

```
$ kubectl expose deploy/nginx --port 80
service "nginx" exposed

$ kubectl describe svc/nginx
Name:                   nginx
Namespace:              default
Labels:                 run=nginx
Annotations:            <none>
Selector:               run=nginx
Type:                   ClusterIP
IP:                     10.0.0.143
Port:                   <unset> 80/TCP
Endpoints:              172.17.0.5:80,172.17.0.7:80
Session Affinity:       None
Events:                 <none>
```

这时，查询所有服务就可以看到该服务对象：

```
$ kubectl get svc | grep nginx
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
nginx        ClusterIP   10.109.24.56   <none>        80/TCP    2s
```

讨论

如果想用浏览器访问该服务，那么可以在另外一个终端运行代理，如下所示：

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

然后在浏览器中打开如下地址：

```
$ open http://localhost:8001/api/v1/proxy/namespaces/default/services/nginx/
```

如果想给 nginx 部署手动写一个同样的 Service 对象，那么可以按照以下内容创建 YAML 文件：

```
apiVersion:  v1
kind:        Service
metadata:
  name:      nginx
spec:
  selector:
    run:     nginx
  ports:
  - port:    80
```

需要注意一点，该 YAML 文件中的 selector 可以选中所有构成这一微服务抽象层的 pod。Kubernetes 使用 Service 对象自动配置所有节点上的 iptables，从而使节点能够访问构成该微服务的容器。该 selector 是一个标签查询（请参阅 6.6 节），它返回一组访问点。

如果服务运行出现问题，请检查 selector 中使用的标签，并确认 kubectl get endpoints 可以返回一组访问点。如果不能，则很可能是 selector 没有找到可以匹配的 pod。

pod 的监控程序（如部署、副本控制器等）可以直接操作服务。监控程序与服务可以使用标签找到所需的 pod，但是它们的职能不同：监控程序负责管理 pod 的健康并负责重启 pod，而服务则负责提供可靠的访问渠道。

请参阅

- Kubernetes 服务的相关文档（*https://kubernetes.io/docs/concepts/services-networking/service/*）。
- Kubernetes 教程“使用服务公开应用程序”（*https://kubernetes.io/docs/tutorials/kubernetes-basics/expose-intro/*）。

5.2 验证服务的 DNS 注册项

问题

在创建服务（请参阅 5.1 节）以后，如何验证服务是否成功注册了 DNS？

解决方案

默认情况下，Kubernetes 的服务类型是 ClusterIP，并通过集群的内部 IP 公布服务。如果 DNS 集群插件正常工作，那么可以通过全称域名（Fully Qualified Domain Name，FQDN），以 `$SERVICENAME.$NAMESPACE.svc.cluster.local` 的形式访问该服务。

我们可以在集群的容器内通过交互式 shell，来验证服务是否正常工作。最简便的方法是使用 `kubectl run` 运行 `busybox` 映像，命令如下：

```
$ kubectl run busybox --image busybox -it -- /bin/sh
If you don't see a command prompt, try pressing enter.

/ # nslookup nginx
Server:    10.96.0.10
```


```
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      nginx
Address 1: 10.109.24.56 nginx.default.svc.cluster.local
```

以上命令返回的服务 IP 地址是它的集群 IP 地址。

5.3 改变服务类型

问题

如果已有一个服务，其类型为 5.2 节中介绍的 ClusterIP 类型，如何改变其类型，使之以 NodePort 类型发布应用程序？或者以 LoadBalancer 的服务类型公开云提供商的负载均衡器？

解决方案

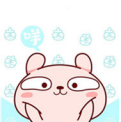
可以使用 kubectl edit，用喜欢的编辑器改变服务的类型。假设清单文件叫做 *simple-nginx-svc.yaml*，其内容如下：

```
kind:       Service
apiVersion: v1
metadata:
  name:     webserver
spec:
  ports:
  - port:   80
  selector:
    app:    nginx
```

创建该 webserver 并查询详细内容的命令如下所示：

```
$ kubectl create -f simple-nginx-svc.yaml

$ kubectl get svc/webserver
NAME        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
webserver   10.0.0.39    <none>        80/TCP    56s
```

接下来，将服务类型改变为 NodePort，如下所示：



```
$ kubectl edit svc/webserver
```

该命令将从 API 服务器中下载该服务当前的规格，并在默认的编辑器中打开，结果如图 5-2 所示（图中编辑器的设定为 EDITOR=vi）。

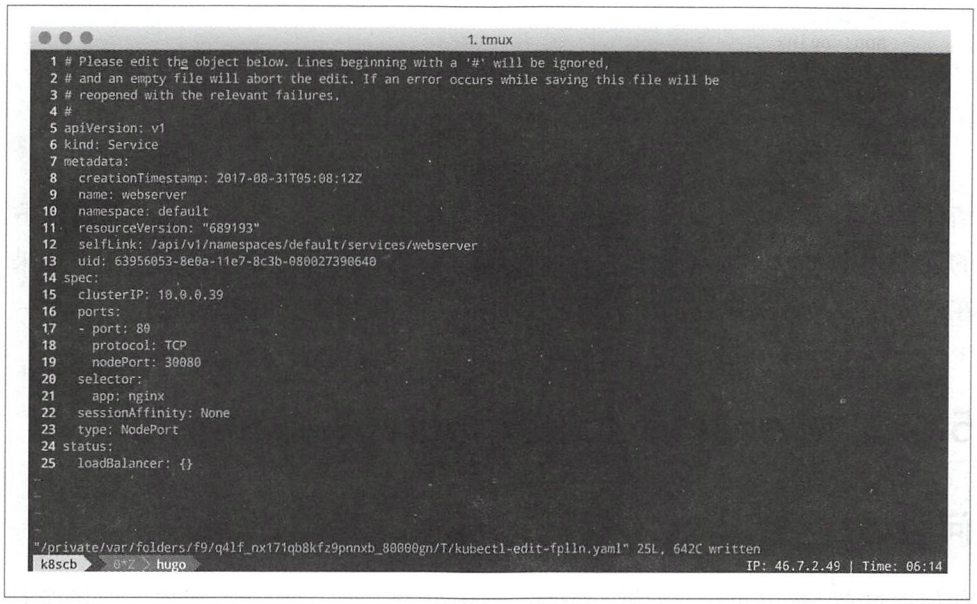


图 5-2：使用 kubectl edit 编辑服务的截图

在保存编辑（type 改为 NodePort，containerPort 改为 node Port）之后，可以通过下面的命令确认变更后的服务：

```
$ kubectl get svc/webserver
NAME        CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
webserver   10.0.0.39    <nodes>       80:30080/TCP   7m

$ kubectl get svc/webserver -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2017-08-31T05:08:12Z
  name: webserver
  namespace: default
  resourceVersion: "689727"
  selfLink: /api/v1/namespaces/default/services/webserver
  uid: 63956053-8e0a-11e7-8c3b-080027390640
spec:
```

```
    clusterIP: 10.0.0.39
    externalTrafficPolicy: Cluster
    ports:
    - nodePort: 30080
      port: 80
      protocol: TCP
      targetPort: 80
    selector:
      app: nginx
    sessionAffinity: None
    type: NodePort
  status:
    loadBalancer: {}
```

请注意，服务类型可以根据实际情况随意修改。但是，某些类型可能有隐藏的规则，比如 `loadBalancer`，可能会触发公共云基础设施组件的配置，如果在不知情并（或）没有监控的情况下使用，可能会造成昂贵的开销。

5.4 在 Minikube 上配置 ingress controller

问题

如何通过在 Minikube 上配置 ingress controller 了解 `ingress` 对象？你可能对 `ingress` 对象很感兴趣，因为它可以帮助我们从 Kubernetes 集群的外部访问 Kubernetes 上运行的应用程序，而无需创建 `NodePort` 或 `LoadBalancer` 类型的服务。

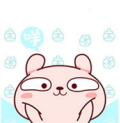
解决方案

你需要部署 ingress 控制器，才能让 `Ingress` 对象（请参阅 5.5 节）生效并提供从集群外部访问 pod 的路由。

在 Minikube 上，激活 ingress 插件的命令如下：

```
$ minikube addons enable ingress
```

运行完命令后，应该可以在 Minikube 插件列表中看到 ingress。查看命令如下所示：



```
$ minikube addons list | grep ingress
- ingress: enabled
```

大约 1 分钟后，kube-system 命名空间中将出现两个新的 pod：

```
$ kubectl get pods -n kube-system
NAME                              READY     STATUS    RESTARTS   AGE
default-http-backend-6tv69        1/1       Running   1          1d
...
nginx-ingress-controller-95dqr    1/1       Running   1          1d
...
```

现在可以创建 Ingress 对象了。

请参阅

- Ingress 的文档（*https://kubernetes.io/docs/concepts/services-networking/ingress/*）。
- 基于 nginx 的 ingress 控制器源代码（*https://github.com/kubernetes/ingress-nginx/blob/master/README.md*）。

5.5 从集群外部访问服务

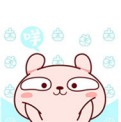
问题

如何从集群外部访问 Kubernetes 的服务？

解决方案

可以使用 ingress 控制器（请参阅 5.4 节），你可以通过创建 5ingress 对象配置 ingress 控制器。Ingress 规则（配置 nginx 服务路径）的清单文件如下所示：

```
$ cat nginx-ingress.yaml
kind:                                   Ingress
apiVersion:                             extensions/v1beta1
metadata:
```

```
    name:                                   nginx-public
    annotations:

      ingress.kubernetes.io/rewrite-target: /
  spec:
    rules:
    - host:
      http:
        paths:
        - path:                             /web
          backend:
            serviceName:                    nginx
            servicePort:                    80
```

```
$ kubectl create -f nginx-ingress.yaml
```

现在可以从 Kubernetes 的仪表盘上看到为 nginx 创建的 ingress 对象了（见图 5-3）。

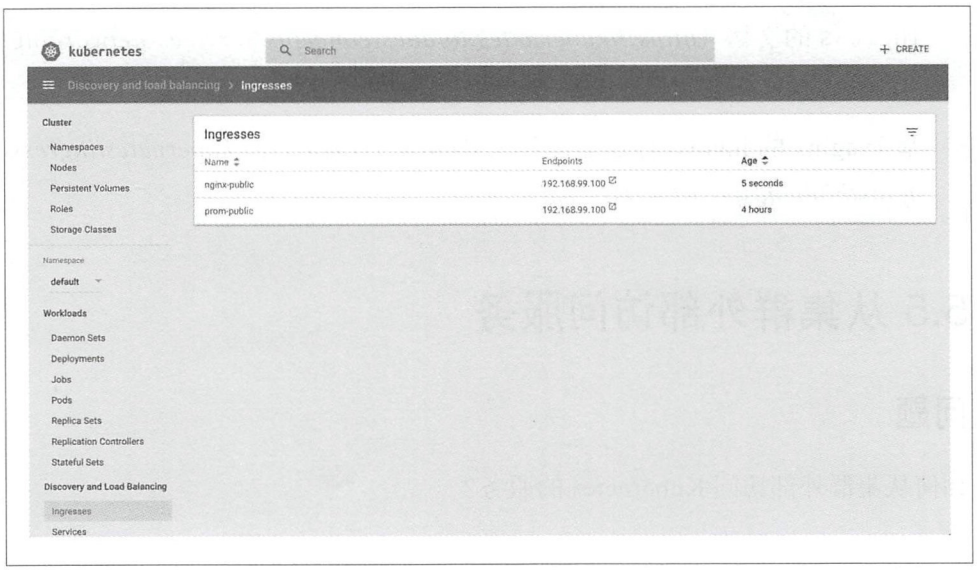


图 5-3：nginx 的 ingress 对象的截图

从 Kubernetes 的仪表盘上，我们看到可以通过 IP 地址 192.168.99.100 访问 nginx，并且清单文件定义了它的公开访问路径为 /web。掌握了这些信息后，我们就可以从集群外部访问 nginx 了，如下所示：

```
$ curl -k https://192.168.99.100/web
<!DOCTYPE html>
<html>
```

```
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;

        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

讨论

一般来说，ingress 的工作方式如图 5-4 所示。ingress 控制器监听 API 服务器的 /ingresses 访问点，并读取新规则。然后配置路由，将外部的访问分配到具体的（集群内部）服务，如图 5-4 中端口 9876 上的服务 1。

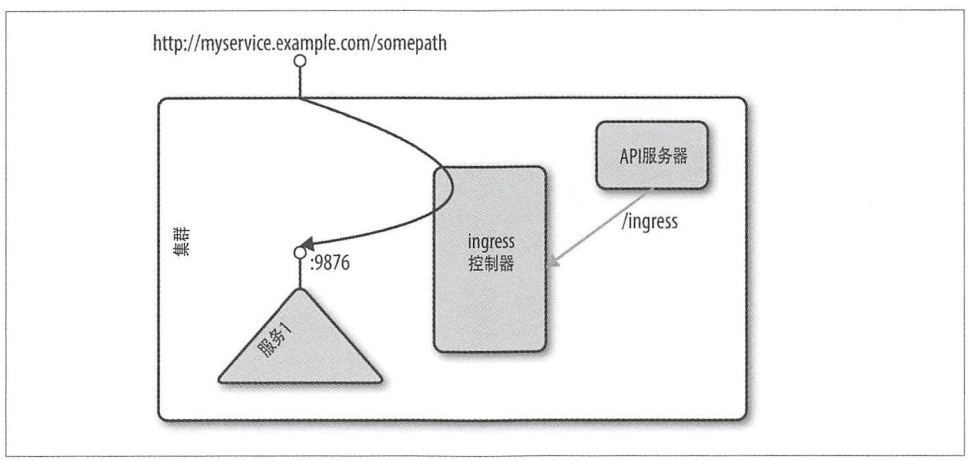
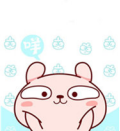


图 5-4：Ingress 的概念

本节中我们使用了 Minishift，它提供的 ingress 控制器插件十分好用。通常，你需要安装 ingress 控制器。请参照 GitHub 上的说明（*https://github.com/kubernetes/ingress-nginx*）。

请参阅

- GitHUb 上 kubernetes/ingress-nginx 代码库（*https://github.com/kubernetes/ingress-nginx*）。
- Milos Gajdos 的博文“Kubernetes 服务与 X 光下的 Ingress”（*http://containerops.org/2017/01/30/kubernetes-services-and-ingress-under-x-ray/*）。
- Daemonza 的博文“Kubernetes 的 nginx-ingress 控制器”（*https://daemonza.github.io/2017/02/13/kubernetes-nginx-ingress-controller/*）。



第 6 章

探索 Kubernetes 的 API 与关键元数据

本章中，我们将通过各小节介绍与 Kubernetes 对象以及 API 的基本交互。不论是命名空间中的对象（比如部署等），还是全集群有效的对象（比如节点等），Kubernetes 的每个对象都有各自的字段，例如 `metadata`（元数据）、`spec`（规格）与 `status`（状态）[注 1]。Kubernetes API 服务器管理的 spec 描述了对象应有的状态，而 `status` 保存了对象的真实状态。

6.1 发现 Kubernetes 上 API 的访问点

问题

如何找出 Kubernetes API 服务器上的各个 API 访问点？

解决方案

如果可以通过无需认证的私有端口访问 API 服务器，那么可以直接向 API 服

注 1：请参阅文档：Kubernetes，“Understanding Kubernetes Objects”（*https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/*）。

务器发出 HTTP 请求，并探索各个访问点。例如，通过 Minikube 可以在虚拟机内部使用 ssh（minikube ssh），并通过端口 8080 访问 API 服务器，如下所示：

```
$ curl localhost:8080/api/v1
...
{
      "name": "pods",
      "namespaced": true,
      "kind": "Pod",
      "verbs": [
        "create",
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "proxy",
        "update",
        "watch"
      ],
      "shortNames": [
        "po"
      ]
    },
...
```

以上给出了一个 Pod 类型对象的实例，以及对象允许的操作，例如：get 和 delete 等。

如果无法直接访问机器上运行的 Kubernetes API 服务器，那么可以使用 kubectl 在本地代理 API。如此一来，就可以访问本地的 API 服务器了，但是需要认证：

```
$ kubectl proxy --port=8001 --api-prefix=/
```

在另外一个窗口中，运行下面的命令：

```
$ curl localhost:8001/foobar
```

使用 /foobar 的 API 路径可以查看所有该 API 上的访问点。请注意 --port 和 --api-prefix 为可选参数。

讨论

在查看 API 的时候，常常会看到不同的访问点，例如：

- `/api/v1`
- `/apis/apps`
- `/apis/authentication.k8s.io`
- `/apis/authorization.k8s.io`
- `/apis/autoscaling`
- `/apis/batch`

每个访问点都对应一个 API 组。在同一个组内，API 对象通过版本控制（比如：`v1beta1`、`v1beta2`）来显示对象的成熟度。例如，Pod、服务、配置映射和加密信息都属于 `/api/v1` 的 API 组，而部署则属于 `/apis/extensions/v1beta1` 的 API 组。

对象所属的组是对象规格中定义的 `apiVersion` 的一部分，请查阅如下 API 参考文档（*https://v1-7.docs.kubernetes.io/docs/reference/*）。

请参阅

- Kubernetes API 概 览（*https://kubernetes.io/docs/reference/api-overview/*）。
- Kubernetes API 规范（*https://github.com/kubernetes/community/blob/master/contributors/devel/api-conventions.md*）。

6.2 掌握 Kubernetes 清单文件的结构

问题

尽管 Kubernetes 拥有非常方便的 `kubectl run` 和 `kubectl create` 可以生成

对象，但是你需要了解如何编写 Kubernetes 清单文件，以表述 Kubernetes 的对象规格。为此，需要掌握清单文件的一般结构。

解决方案

在 6.1 节中，我们学习了各种 API 组，以及如何发现具体的对象属于哪个组。

所有的 API 资源都是对象或列表。所有的资源都有 `kind` 和 `apiVersion` 字段。另外。每个对象的 `kind` 都有 `metadata`。而这个 `metadata` 则包含对象的名称，所属的命名空间（请参阅 6.3 节），以及一些可能的标签（请参阅 6.6 节）和注解（请参阅照 6.7 节）。

举例来说，一个 pod 的 `kind` 值为 Pod，`apiVersion` 的值为 `v1`，那么简单的 YAML 清单文件开头如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
...
```

其次，清单文件中大多数的对象都包含 `spec` 字段，且在创建后还将返回 `status`：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  ...
status:
  ...
```

请参阅

- 掌握 Kubernetes 对象（*https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/*）。

6.3 通过创建命名空间避免命名冲突

问题

如何在创建两个名称相同的对象的时候，避免命名冲突？

解决方案

创建命名空间，并将两个对象放在不同的命名空间之中。

如果没有特殊声明，那么创建的对象将被放入 `default` 命名空间之中。下面我们将尝试创建一个名为 `my-app` 的命名空间，并查看已有的命名空间，如下所示。你可以看到 `default` 命名空间，另个其他两个系统自带的命名空间(`kube-system` 和 `kube-public`)，以及刚才创建的命名空间 *my-app*：

```
$ kubectl create namespace my-app
namespace "my-app" created

$ kubectl get ns
NAME          STATUS    AGE
default       Active    30s
my-app        Active    1s
kube-public   Active    29s
kube-system   Active    30s
```

还有一个方法，你可以自己编写清单文件来创建命名空间。首先请将下列清单文件保存为 *app.yaml*，然后使用 `kubectl create -f app.yaml` 命令创建该命名空间：

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-app
```

讨论

如果在同一个命名空间（比如 `default`）中创建两个名称相同的对象会引发

冲突，Kubernetes API 服务器将返回错误。但是，如果在不同的命名空间中创建第二个对象，那么 API 服务器会成功地完成创建：

```
$ kubectl run foobar --image=ghost:0.9
deployment "foobar" created

$ kubectl run foobar --image=nginx:1.13
Error from server (AlreadyExists): deployments.extensions "foobar" already exists

$ kubectl run foobar --image=nginx:1.13 --namespace foobar
deployment "foobar" created
```

这是因为很多 Kubernetes 的 API 对象都指定了命名空间。各个对象所属的命名空间是对象元数据中的一部分。

Kube-system 是系统管理员专属命名空间，而 kube-public 命名空间则用于保存集群上任何用户的公开对象。

6.4 设置命名空间的配额

问题

如何限定命名空间可访问的资源，例如，限定命名空间中可以运行的 pod 总数？

解决方案

可以使用 ResourceQuota 指定命名空间的限制：

```
$ cat resource-quota-pods.yaml
apiVersion: v1
kind: ResourceQuota
metadata:

  name: podquota
spec:
  hard:
```

```
    pods: "10"

$ kubectl create namespace my-app

$ kubectl create -f resource-quota-pods.yaml --namespace=my-app

$ kubectl describe resourcequota podquota --namespace=my-app
Name:           podquota
Namespace:      my-app
Resource        Used    Hard
--------        ----    ----
pods            0       10
```

讨论

可以为每个命名空间设置一系列的配额，包括但不限于 pod、加密信息和配置映射等。

请参阅

- 设置 API 对象的配额（*https://kubernetes.io/docs/tasks/administer-cluster/quota-api-object/*）。

6.5 给对象贴标签

问题

如何标记对象以方便日后查找？该标签可用于进一步的最终用户查询（请参阅 6.6 节），或用于系统自动化环境。

解决方案

可以使用 `kubectl label` 命令。例如，通过键值 `tier=frontend` 将一个 pod 标记为 `foobar`，如下所示：

```
$ kubectl label pods foobar tier=frontend
```

请通过 kubectl label --help 命令查看标记命令的完整帮助。你可以使用帮助命令找到删除标签、更新标签、以及标记命名空间中所有资源的方法。

讨论

在 Kubernetes 中，用户可以使用标签灵活地组织对象（非分层方式）。标签可以是任何 Kubernetes 预定义之外的键值。换句话说，必须确认键值的内容在系统中没有特殊含义。可以使用标签表述成员关系（比如，对象 X 属于部门 ABC）、环境（比如，该服务在产品环境中运行），或任何需要组织的对象。请注意标签的长度和允许的赋值有限制[注 2]。

6.6 使用标签进行查询

问题

如何有效地查询对象？

解决方案

使用 kubectl get --selector 命令。例如，我们有如下 pod：

```
$ kubectl get pods --show-labels
NAME                    READY   ...   LABELS
cockroachdb-0           1/1     ...   app=cockroachdb,
cockroachdb-1           1/1     ...   app=cockroachdb,
cockroachdb-2           1/1     ...   app=cockroachdb,
jump-1247516000-sz87w   1/1     ...   pod-template-hash=1247516000,run=jump
nginx-4217019353-462mb  1/1     ...   pod-template-hash=4217019353,run=nginx
nginx-4217019353-z3g8d  1/1     ...   pod-template-hash=4217019353,run=nginx
prom-2436944326-pr60g   1/1     ...   app=prom,pod-template-hash=2436944326
```

注 2：请参阅文档：Kubernetes，“Labels and Selectors: Syntax and character set”（*https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#syntax-and-character-set*）。

你可以选出属于 CockroachDB 应用（app=cockroachdb）的 pod：

```
$ kubectl get pods --selector app=cockroachdb
NAME            READY     STATUS    RESTARTS   AGE
cockroachdb-0   1/1       Running   0          17h
cockroachdb-1   1/1       Running   0          17h
cockroachdb-2   1/1       Running   0          17h
```

讨论

标签属于对象元数据的一部分。Kubernetes 中的任何对象都可以被标记。Kubernetes 本身也可以在部署（请参阅 4.1 节）和服务（请参阅第 5 章）中使用标签选择 pod。

可以通过 `kubectl label` 命令（请参阅 6.5 节）手动添加标签，也可以在对象的清单文件中定义标签，如下所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: foobar
  labels:
    tier: frontend
...
```

在添加标签后，可以使用 `kubectl get` 命令进行查看，具体参数如下：

- `-l` 是 `--selector` 的简化形式，可以通过指定的 `key=value` 查询对象。
- `--show-labels` 将显示每个对象返回的所有标签。
- `-L` 将在显示结果中添加一列，用以返回指定标签的值。
- 很多对象的类型都支持集合方式的查询，也就是说可以使用“必须含有 X 和 / 或 Y 标签”的形式执行查询。例如，`kubectl get pods -l 'env in (production, development)'` 将返回产品环境或开发环境中的 pod。

如果有两个 pod，其中一个含有标签 `run=barfoo`，而另外一个标签为 `run=foo bar`，那么下列查询可以返回如下类似的结果：

```
$ kubectl get pods --show-labels
NAME                      READY   ...   LABELS
barfoo-76081199-h3gwx     1/1     ...   pod-template-hash=76081199,run=barfoo
foobar-1123019601-6x9w1   1/1     ...   pod-template-hash=1123019601,run=foobar

$ kubectl get pods -Lrun
NAME                      READY   ...   RUN
barfoo-76081199-h3gwx     1/1     ...   barfoo
foobar-1123019601-6x9w1   1/1     ...   foobar

$ kubectl get pods -l run=foobar
NAME                      READY   ...
foobar-1123019601-6x9w1   1/1     ...
```

请参阅

- Kubernetes 的标签（*https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/*）。

6.7 通过命令注解资源

问题

如何给一个资源加注解？注解的内容可以是通用的、非识别性键值，而且可能包含人类不可读的数据。

解决方案

可以使用 kubectl annotate 方法：

```
$ kubectl annotate pods foobar \
description='something that you can use for automation'
```

讨论

一般注解会用于增加 Kubernetes 的自动化。例如，当使用 kubectl run 命令创建部署的时候，如果忘了使用 --record 参数，那么在历史记录（请参阅 4.5

节）中我们将看到 change-cause 一列为空。从 Kubernetes v1.6.0 开始，为了记录引发部署变更的命令，可以使用 kuber netes.io/change-cause 键对其进行注释。对于上述 foobar 部署，添加注解的命令如下：

```
$ kubectl annotate deployment foobar \
kubernetes.io/change-cause="Reason for creating a new revision"
```

部署的后续变更都将记录在册。

第 7 章

管理具体的工作负载

在第 4 章中，我们介绍了如何启动长期运行的应用程序，例如网络服务器或应用服务器。在本章中，我们将更加具体地讨论工作负荷。例如，启动终止批处理等进程，在特定节点上运行 pod，以及管理有状态和不在云端的原生应用等。

7.1 运行批处理

问题

如何运行一个需要较长时间才能运行完成的进程，例如批量转换、备份操作、或数据库模式的升级等？

解决方案

使用 Kubernetes 的 job 资源可以启动并监视执行批处理的 pod[注 1]。

首先，在一个名为 *counter-batchjob.yaml* 的文件中定义该 job 的清单文件：

```
apiVersion:          batch/v1
kind:                Job
```

注 1：请参阅文档：Kubernetes，“Jobs - Run to Completion”（*https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/*）。

```
metadata:
  name:             counter
spec:
  template:

    metadata:
      name:         counter
    spec:
      containers:
      - name:       counter
        image:      busybox
        command:
         - "sh"
         - "-c"
         - "for i in 1 2 3 ; do echo $i ; done"
      restartPolicy: Never
```

然后启动该 job 并查看它的状态：

```
$ kubectl create -f counter-batch-job.yaml
job "counter" created

$ kubectl get jobs
NAME      DESIRED   SUCCESSFUL   AGE
counter   1         1            22s

$ kubectl describe jobs/counter
Name:           counter
Namespace:      default
Selector:       controller-uid=634b9015-7f58-11e7-b58a-080027390640
Labels:         controller-uid=634b9015-7f58-11e7-b58a-080027390640
                job-name=counter
Annotations:    <none>
Parallelism:    1
Completions:    1
Start Time:     Sat, 12 Aug 2017 13:18:45 +0100
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:       controller-uid=634b9015-7f58-11e7-b58a-080027390640
                job-name=counter
  Containers:
   counter:
    Image:      busybox
    Port:       <none>
    Command:
      sh
      -c
      for i in 1 2 3 ; do echo $i ; done
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen  ...  ...  ...  Type    Reason            Message
```

```
---------  ...  ...  ...  ------  ------            -------
31s        ...  ...  ...  Normal  SuccessfulCreate  Created pod: counter-0pt20
```

最后，确认 job 是否已运行（从 1 数到 3）：

```
$ kubectl logs jobs/counter
1
2
3
```

如上所示，counter job 按照预期完成运行。

如果不再需要该 job 的时候，可以使用 kubectl delete jobs/counter 将其删除。

7.2 在 Pod 内按照计划时间运行任务

问题

如何在 Kubernetes 管理的 pod 内按照具体的计划时间运行任务？

解决方案

可以使用 Kubernetes 的 CronJob 对象。CronJob 对象继承了更加通用的 Job 对象（请参阅 7.1 节）。

你可以仿照下面的例子编写清单文件来计划定期的 job。spec 中包含 schedule 一节，其采用的是 crontab 格式。template 一节描述了需要运行的 pod 以及需要执行命令（每个小时它都会在 stdout 中输出当前的日期和时间）：

```
apiVersion:              batch/v2alpha1
kind:                    CronJob
metadata:
  name:                  hourly-date
spec:
  schedule:              "0 * * * *"
  jobTemplate:
    spec:
```

```
    template:
      spec:
        containers:
        - name:         date
          image:        busybox
          command:
            - "sh"
            - "-c"
            - "date"
        restartPolicy: OnFailure
```

请参阅

- CronJob 文档（*https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/*）。

7.3 在每个节点上运行基础设施的服务

问题

如何启动日志收集器或监视代理等基础设施的服务，并确保每个节点上都恰好运行了一个 pod？

解决方案

使用 DaemonSet 启动并监控服务进程。例如，为了在集群的每个节点上启动 Fluentd，你可以仿照下面的例子编写一个 *fluentddaemonset.yaml* 文件：

```
kind:                DaemonSet
apiVersion:          extensions/v1beta1
metadata:
  name:              fluentd
spec:
  template:
    metadata:
      labels:
        app:         fluentd
      name:          fluentd
    spec:
      containers:
```

```
      - name:          fluentd
        image:         gcr.io/google_containers/fluentd-elasticsearch:1.3
        env:
         - name:       FLUENTD_ARGS
           value:      -qq
        volumeMounts:
         - name:       varlog
           mountPath:  /varlog
         - name:       containers
           mountPath:  /var/lib/docker/containers
      volumes:
         - hostPath:
             path:     /var/log
           name:       varlog
         - hostPath:
             path:     /var/lib/docker/containers
           name:       containers
```

然后按照下面的方法启动 DaemonSet：

```
$ kubectl create -f fluentd-daemonset.yaml
daemonset "fluentd" created

$ kubectl get ds
NAME     DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE-SELECTOR  AGE
fluentd  1        1        1      1           1          <none>         17s

$ kubectl describe ds/fluentd
Name:           fluentd
Selector:       app=fluentd
Node-Selector:  <none>
Labels:         app=fluentd
Annotations:    <none>
Desired Number of Nodes Scheduled: 1
Current Number of Nodes Scheduled: 1
Number of Nodes Scheduled with Up-to-date Pods: 1
Number of Nodes Scheduled with Available Pods: 1
Number of Nodes Misscheduled: 0
Pods Status:    1 Running / 0 Waiting / 0 Succeeded / 0 Failed
...
```

讨论

请注意在上述输出中，由于该命令是在 Minikube 上运行的，所以你仅可以看到一个运行的 pod，因为这里仅设置了一个节点。如果集群中有 15 个节点的话，那么每个节点都有 1 个 pod 运行，所以一共可以看到 15 个 pod。还可以在 DaemonSet 清单文件的 spec 中使用 nodeSelector 一节来限定某些节点上的服务。

7.4 管理有状态的主从应用

问题

如何运行一个要求各个 pod 具有不同特征的应用程序，例如数据库应用，它包括一个处理读写的主应用，和几个只读的从应用？这种情况下无法使用部署，因为部署只能像放牛一样监控一群相同的 pod，而这里你需要一个监控者，可以像对待宠物那样分别处理多个 pod。

解决方案

可以使用 `StatefulSet`，它可以使工作负荷拥有唯一网络名，允许优雅地进行部署、伸缩和终止，或启用永久性存储。例如，为了运行流行的可伸缩数据库 `CockroachDB`，可以使用 Kubernetes 提供的例子[注2]，其核心包含了如下 `StatefulSet`：

```
apiVersion: apps/v1beta1
kind:                       StatefulSet
metadata:
  name:                     cockroachdb
spec:
  serviceName:              "cockroachdb"
  replicas:                 3
  template:
    metadata:
      labels:
        app:                cockroachdb
    spec:
      initContainers:
      - name:               bootstrap
        image:              cockroachdb/cockroach-k8s-init:0.2
        imagePullPolicy:    IfNotPresent
        args:
        - "-on-start=/on-start.sh"
        - "-service=cockroachdb"
        env:
        - name:             POD_NAMESPACE
```

注 2：请参阅：Kubernetes 在 GitHub 上给出的 cockroachdb 的例子 *cockroachdb-statefulset.yaml*（*https://github.com/kubernetes/kubernetes/blob/master/examples/cockroachdb/cockroachdb-statefulset.yaml*）。

```
          valueFrom:
            fieldRef:
              fieldPath:      metadata.namespace
        volumeMounts:
        - name: datadir
          mountPath:          "/cockroach/cockroach-data"
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator:  In
                  values:
                  - cockroachdb
              topologyKey:   kubernetes.io/hostname
      containers:
      - name:                cockroachdb
        image:               cockroachdb/cockroach:v1.0.3
        imagePullPolicy:     IfNotPresent
        ports:
        - containerPort:     26257
          name: grpc

        - containerPort:     8080
          name: http
        volumeMounts:
        - name: datadir
          mountPath:         /cockroach/cockroach-data
        command:
          - "/bin/bash"
          - "-ecx"
          - |
            if [ ! "$(hostname)" == "cockroachdb-0" ] || \
               [ -e "/cockroach/cockroach-data/cluster_exists_marker" ]
            then
              CRARGS+=("--join" "cockroachdb-public")
            fi
            exec /cockroach/cockroach ${CRARGS[*]}
      terminationGracePeriodSeconds: 60
      volumes:
      - name:                datadir
        persistentVolumeClaim:
          claimName:         datadir
  volumeClaimTemplates:
  - metadata:
      name:                  datadir
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
    spec:
      accessModes:
        - "ReadWriteOnce"
      resources:
```

```
      requests:
        storage:          1Gi
```

通过以下命令运行该文件：

```
$ curl -s -o cockroachdb-statefulset.yaml \
            https://raw.githubusercontent.com/kubernetes/kubernetes/master/ \
            examples/cockroachdb/cockroachdb-statefulset.yaml

$ curl -s -o crex.sh \
            https://raw.githubusercontent.com/kubernetes/kubernetes/master/ \
            examples/cockroachdb/minikube.sh

$ ./crex.sh
+ kubectl delete statefulsets,persistentvolumes,persistentvolumeclaims,services...
...
+ kubectl create -f -
persistentvolumeclaim "datadir-cockroachdb-3" created
+ kubectl create -f cockroachdb-statefulset.yaml
service "cockroachdb-public" created
service "cockroachdb" created
poddisruptionbudget "cockroachdb-budget" created
statefulset "cockroachdb" created
```

现在你可以在 Kubernetes 的仪表盘内看到创建的 StatefulSet 对象以及 pod 了（见图 7-1）。

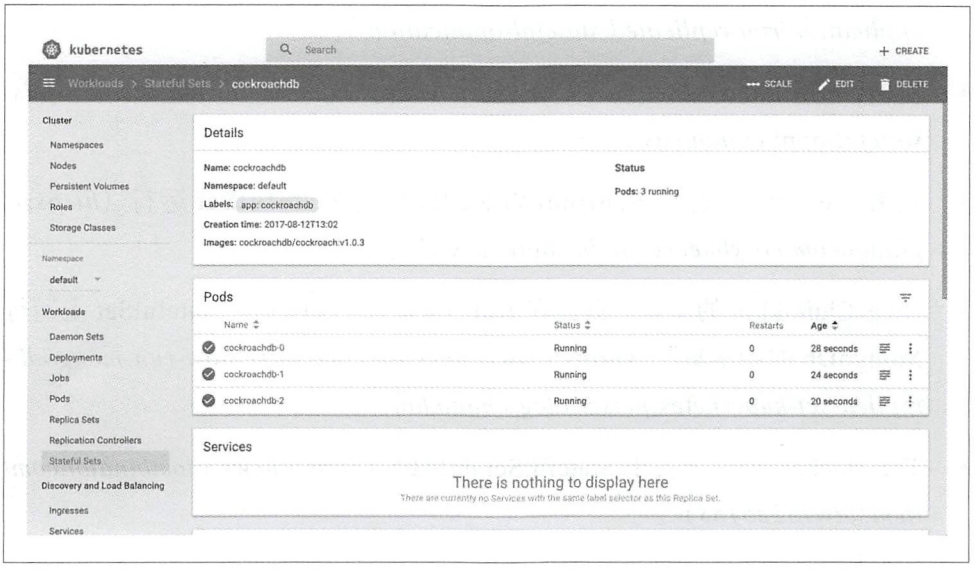


图 7-1：StatefulSet 的截图

讨论

最初，Kubernets 的 StatefulSet 被称作 PetSet。你可能猜到了如此命名的动机。StatefulSet 在 Kubernetes 1.7 中成为了 beta 功能，这意味着这个 API 不会再有任何改变，只会有界面上的修改。StatefulSet 是一个控制器，为其监控的 pods 提供唯一的身份。请注意，出于安全的考虑，删除 StatefulSet 的时候并不会删除与之相关联的卷。

StatefulSet 的另一常见的应用场景是运行那些并非刻意为 Kuberntes 编写的应用。从 Kuberntes 的角度看来，这类应用有时被称作遗留的应用。在后续章节中我们把这类应用看作非云端的原生应用。StatefulSet 很适合监控这类应用。

请参阅

- StatefulSet 的基本知识（*https://kubernetes.io/docs/tutorials/stateful-application/basic-stateful-set/*）。
- 运行一个复制的有状态的应用程序（*https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/*）。
- 实例：使用 StatefulSet 部署 Cassandra（*https://kubernetes.io/docs/tutorials/stateful-application/cassandra/*）。
- 在 Kubernets 上通过 Sentinel 将 Redis 作为 StatefulSet 运行（*https://github.com/corybuecker/redis-stateful-set*）。
- Oleg Chunikhin 的文章“如何在 Kubernetes 的 Petset 或 StatefulSet 上运行 MongoDb 的副本集”（*https://www.linkedin.com/pulse/how-run-mongodb-replica-set-kubernetes-petset-oleg-chunikhin*）。
- The Hacker New 上关于 StatefulSet 的讨论（*https://news.ycombinator.com/item?id=13225183*）。

7.5 影响 Pod 的启动行为

问题

Pod 依赖于其他服务才能正常运行?

解决方案

通过初始化容器来影响 pod 的启动行为。

假设你想启动一个依赖于后台服务提供内容的 nginx 网络服务器。那么你需要确认只有在后台服务启动并运行的时候才能启动该 nginx pod。

首先，让我们创建该网络服务依赖的后台服务：

```
$ kubectl run backend --image=mhausenblas/simpleservice:0.5.0
deployment "backend" created

$ kubectl expose deployment backend --port=80 --target-port=9876
```

接下来使用如下清单文件 *nginx-init-container.yaml*，来启动 nginx 实例，并确认只有在 backend 部署提供数据的时候才能正常启动：

```
kind:                Deployment
apiVersion:          apps/v1beta1
metadata:
  name:              nginx
spec:
  replicas:          1
  template:
    metadata:
      labels:
        app:         nginx
    spec:
      containers:
      - name:        webserver
        image:       nginx
        ports:
```

```
    - containerPort: 80
  initContainers:
  - name:         checkbackend

    image:        busybox
    command:      ['sh', '-c', 'until nslookup backend.default.svc; do echo
                   "Waiting for backend to come up"; sleep 3; done; echo
                   "Backend is up, ready to launch web server"']
```

现在你可以启动 nginx 部署，并通过查看其监控的 pod 的日志，确认初始化容器是否真正起作用了：

```
$ kubectl create -f nginx-init-container.yaml
deployment "nginx" created

$ kubectl get po
NAME                       READY     STATUS    RESTARTS   AGE
backend-853383893-2g0gs    1/1       Running   0          43m
nginx-2101406530-jwghn     1/1       Running   0          10m

$ kubectl logs nginx-2101406530-jwghn -c checkbackend
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      backend.default.svc
Address 1: 10.0.0.46 backend.default.svc.cluster.local
Backend is up, ready to launch web server
```

如上所示，初始化容器中的命令确实按照计划执行了。

第 8 章

卷与配置数据

在 Kubernetes 中，卷（Volume）指的是 pod 上运行的所有容器都可以访问的目录，卷可以保证个别容器重启的时候，数据依然可以保存下来。

根据卷的作用与字面呈现的含义，我们可以将卷划分为以下几类：

- 节点本地卷，比如：`emptyDir` 或 `hostPath`。
- 通用网络卷，比如：`nfs`、`glusterfs` 或 `cephfs`。
- 云服务商的卷，比如：`awsElasticBlockStore`、`azureDisk` 或 `gcePersistentDisk`。
- 特殊用途的卷，比如：`secret` 或 `gitRepo`。

选择哪种卷完全取决于实际的使用情况。例如，对于临时的暂存空间，`emptyDir` 就可以了，但是如果需要确保在节点故障的时候数据仍然可以永久保存下来，那么可能需要使用网络上的卷，或者如果 Kubernetes 在公共云端上运行的话，则可以使用云服务商的卷。

8.1 通过本地卷在容器间交换数据

问题

如果 pod 上运行了两个或多个容器，如何通过文件系统的操作交换数据？

解决方案

可以使用 emptyDir 类型的本地数据卷。

首先准备好如下的 pod 清单文件 *exchangedata.yaml*，其中包含了两个容器（c1 和 c2），两者通过不同的挂载点，在各自的文件系统中挂载了本地数据卷 xchange：

```
apiVersion:          v1
kind:                Pod
metadata:
  name:              sharevol
spec:
  containers:
  - name:            c1
    image:           centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name:        xchange
        mountPath:   "/tmp/xchange"
  - name:            c2
    image:           centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name:        xchange
        mountPath:   "/tmp/data"
  volumes:
  - name:            xchange
    emptyDir:        {}
```

现在启动该 pod，运行 exec 命令进入 pod，从一个容器上创建数据，然后从另外一个上读取：

```
$ kubectl create -f exchangedata.yaml
pod "sharevol" created

$ kubectl exec sharevol -c c1 -i -t -- bash
[root@sharevol /]# mount | grep xchange
/dev/vda1 on /tmp/xchange type ext4 (rw,relatime,data=ordered)
[root@sharevol /]# echo 'some data' > /tmp/xchange/data
[root@sharevol /]# exit
```

```
$ kubectl exec sharevol -c c2 -i -t -- bash
[root@sharevol /]# mount | grep /tmp/data
/dev/vda1 on /tmp/data type ext4 (rw,relatime,data=ordered)

[root@sharevol /]# cat /tmp/data/data
some data
```

讨论

本地数据卷由 pod 及其容器所处的节点提供。如果节点关闭或需要做节点维护（请参阅 12.8 节），那么本地数据卷就会消失，并且所有的数据也会丢失。

有些情况下可以使用本地数据卷，例如对于一些暂存空间，或者当正确数据来自于其他地方（如 S3 bucket 等），而本地数据卷仅用作缓存的时候。大多数情况下，由网络存储提供的卷更为常用（请参阅 8.6 节）。

请参阅

- 关于 Kubernetes 卷的文档（*https://kubernetes.io/docs/concepts/storage/volumes/*）。

8.2 通过 Secret 类型的卷将 API 的访问密钥传递给 pod

问题

管理员如何用安全的方式将 API 访问密钥提供给开发者？也就是说，不将秘钥以明文的方式写在 Kubernetes 的清单文件中。

解决方案

可以使用 `secret` 类型的本地数据卷。

假设你希望让开发人员通过密码 open sesame 访问一个外部服务。

首先，将密码保存到一个叫做 *passphrase* 的文本文件中：

```
$ echo -n "open sesame" > ./passphrase
```

然后，使用该 *passphrase* 文件创建 secret：

```
$ kubectl create secret generic pp --from-file=./passphrase
secret "pp" created

$ kubectl describe secrets/pp
Name:           pp
Namespace:      default
Labels:         <none>
Annotations:    <none>

Type:   Opaque


Data
====
passphrase:     11 bytes
```

从管理员的角度来说，做完这些工作就足够了，接下来开发人员可以使用该 secret 了。现在让我们转换一下身份，假设你是开发人员，你想在 pod 内使用该密码。

例如，在使用该 secret 的时候，可以将其作为一个卷挂载到 pod 上，然后当成一个普通文件读取。如下所示创建 pod 并挂载卷：

```
apiVersion:         v1
kind:               Pod
metadata:
  name:             ppconsumer
spec:
  containers:
  - name:           shell
    image:          busybox
    command:
      - "sh"
      - "-c"
      - "mount | grep access  && sleep 3600"
    volumeMounts:
      - name:       passphrase
        mountPath: "/tmp/access"
```

```
        readOnly:  true
  volumes:
  - name:          passphrase
    secret:
      secretName:  pp
```

现在启动 pod，看看它的日志，我们希望可以看到该 pp 密码文件已经挂载为 */tmp/access/passphrase*：

```
$ kubectl create -f ppconsumer.yaml
pod "ppconsumer" created

$ kubectl logs ppconsumer
tmpfs on /tmp/access type tmpfs (ro,relatime)
```

这时如果想要在运行的容器内访问该密码，只需简单地读取 */tmp/access* 中的 *passphrase* 文件，如下所示：

```
$ kubectl exec ppconsumer -i -t -- sh

/ # cat /tmp/access/passphrase
open sesame
```

讨论

Secret 存在于命名空间的环境中，所以在设置和使用它们的时候，需要考虑到这一点。

通过以下的方式，可以在 pod 上运行的容器内访问 secret：

- 卷（如前面解决方案中所介绍的，加密信息保存在 tmpfs 的卷中）。
- 环境变量。

另外，请注意加密文件最大可容纳 1MB 数据。

除了用户自定义的 secret 之外，Kubernetes 还会自动为访问 API 的服务账号生成 secret。例如，如果安装了 Prometheus（请参阅 11.6 节），那么可以在 Kubernetes 的仪表盘中看到图 8-1 所示的内容。

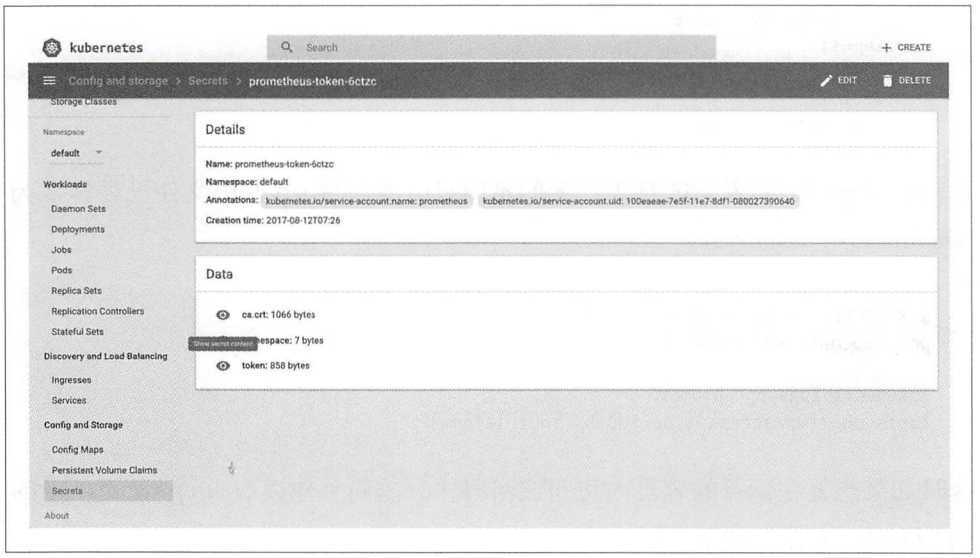


图 8-1. Prometheus 服务账号 secret 的截图

`kubectl create secret` 可以创建三种类型的 secret，你可以根据需要选用最适合的：

- `docker-registry` 类型可以用于注册 Docker。
- 解决方案中所介绍的是 `generic` 类型，它可以根据本地的文件、目录或常量值生成 secret（你需要自行对其进行 base64 编码）。
- `tls` 类型可以用于创建 ingress 的安全 SSL 证书等。

`kubectl describe` 不会用明文显示 secret 的内容。这样可以避免有人偷窥密码。但是，secret 并未经过加密处理，只进行了 base64 编码，所以手动解码很容易：

```
$ kubectl get secret pp -o yaml | \
  grep passphrase | \
  cut -d":" -f 2 | \
  awk '{$1=$1};1' | \
  base64 --decode
open sesame
```

在上述命令中，第一行从 YAML 文件中读取 secret，第二行用 `grep` 找出 `passphrase` 一行：`b3BlbiBzZXNhbWU=`（请注意开头的空格）。然后 `cut` 命令从 `passphrase` 抽出内容，并通过 `awk` 命令去掉开头的空格。最后，用 `base64` 命令将数据还原。

Kubernetes 1.7 之前的版本中，API 服务器将 secret 以明文的方式保存在 `etcd` 中。现在你可以在启动 kube-apiserver 的时候，使用 `--experimental-encryption-provider-config` 参数对 `secret` 进行加密。

请参阅

- Kubernetes 的 secret 文档（*https://kubernetes.io/docs/concepts/configuration/secret/*）。
- 用 Rest 格式加密私密数据（*https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/*）。

8.3 提供配置数据给应用程序

问题

如何提供配置数据给应用程序，而无需将数据存储在容器映像中，或写死在 pod 规格中？

解决方案

使用配置映射。这是 Kubernetes 的一级资源，有了它，你就可以使用环境变量或文件提供配置数据给 pod。

假设我们创建一个配置，其键为 `siseversion`，值为 0.9。这个命令非常简单，如下所示：

```
$ kubectl create configmap siseconfig --from-literal=siseversion=0.9
configmap "siseconfig" created
```

现在可以在部署中使用该配置映射了，假设清单文件 *cmapp.yaml* 包含了如下内容：

```
apiVersion:                 extensions/v1beta1
kind:                       Deployment
metadata:
  name:                     cmapp
spec:
  replicas:                 1
  template:
    metadata:
      labels:
        app:                cmapp
    spec:
      containers:
      - name:               sise
        image:              mhausenblas/simpleservice:0.5.0
        ports:
        - containerPort:    9876
        env:
        - name:             SIMPLE_SERVICE_VERSION
          valueFrom:
            configMapKeyRef:
              name:         siseconfig
              key:          siseversion
```

以上我们演示了如何将配置作为环境变量进行传递。然而，你也可以通过卷将其作为文件挂载到 pod 中。

假设有如下的配置文件 *example.cfg*：

```
debug: true
home: ~/abc
```

你可以创建保存该配置文件的配置映射，如下所示：

```
$ kubectl create configmap configfile --from-file=example.cfg
```

现在可以像使用其他卷一样使用配置映射了。下面是一个 pod 的清单文件，

名为 oreilly，它使用 busybox 的映像，功能只是休眠 3600s。在配置文件的 volumes 一节中，有一个叫做 oreilly 的卷，它使用了上面定义的配置映射 configfile。接下来该卷会挂载到容器内的 /oreilly 路径下。然后就可以在 pod 内访问该文件了：

```
apiVersion:         v1
kind:               Pod
metadata:
  name:             oreilly
spec:
  containers:
  - image:          busybox
    command:
      - sleep
      - "3600"
    volumeMounts:
    - mountPath:    /oreilly
      name:         oreilly
    name:           busybox
  volumes:
  - name:           oreilly
    configMap:
      name:         configfile
```

在 pod 创建后，可以确认 *example.cfg* 确实存在：

```
$ kubectl exec -ti oreilly -- ls -l /oreilly
total 0
lrwxrwxrwx   1 root   root   18 Dec 16 19:36 example.cfg -> ..data/example.cfg

$ kubectl exec -ti oreilly -- cat /oreilly/example.cfg
debug: true
home: ~/abc
```

关于如何使用文件创建配置映射的完整例子，请参阅 11.6 节。

请参阅

- 使用 ConfigMap 配置 pod（*https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/*）。

8.4 在 Minikube 内使用持久卷

问题

如何保证容器所用的硬盘上的数据不会丢失，也就是说，如何保证在重启 pod 的时候数据不会丢失？

解决方案

使用持久卷（Persistent Volume，PV）。在 Minikube 的情况下，可以创建 `hostPath` 类型的 PV，并以挂载普通卷同样的方式将 PV 挂载到容器的文件系统内。

首先，在清单文件 *hostpath-pv.yaml* 中定义名为 `hostpathpv` 的 PV：

```
kind:               PersistentVolume
apiVersion:         v1
metadata:
  name:             hostpathpv
  labels:
    type:           local
spec:
  storageClassName: manual
  capacity:
    storage:        1Gi
  accessModes:
  - ReadWriteOnce
  hostPath:
    path:           "/tmp/pvdata"
```

然而，在创建 PV 之前，需要先在节点中创建 */tmp/pvdata* 目录——也就是 Minikube 实体本身。然后可以通过 `minikube ssh` 进入 Kubernetes 集群运行的节点：

```
$ minikube ssh

$ mkdir /tmp/pvdata && \
  echo 'I am content served from a delicious persistent volume' > / \
  tmp/pvdata/index.html

$ cat /tmp/pvdata/index.html
```

```
I am content served from a delicious persistent volume

$ exit
```

现在节点中的目录已经准备就绪，可以利用清单文件 *hostpath-pv.yaml* 创建 PV 了：

```
$ kubectl create -f hostpath-pv.yaml
persistentvolume "hostpathpv" created

$ kubectl get pv

NAME        CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS      ...   ...   ...
hostpathpv  1Gi        RWO           Retain          Available   ...   ...   ...

$ kubectl describe pv/hostpathpv
Name:           hostpathpv
Labels:         type=local
Annotations:    <none>
StorageClass:   manual
Status:         Available
Claim:
Reclaim Policy: Retain
Access Modes:   RWO
Capacity:       1Gi
Message:
Source:
    Type:       HostPath (bare host directory volume)
    Path:       /tmp/pvdata
Events:         <none>
```

到目前为止，你一直在使用管理员的角色完成了上述工作。你在 Kubernetes 集群上定义了 PV，并向开发人员开放。

现在从开发人员的角度来看，你需要在 pod 中使用该 PV。为了使用 PV，首先需要定义持久卷声明 (Persistent Volume Claim，PVC)，顾名思义这个定义是为了声明这个卷的确是个持久卷，它具有一定的特性，比如大小或存储类等。

为了定义 PVC，首先需要创建一个叫做 *pvc.yaml* 的清单文件，申请 200MB 的空间：

```
kind:               PersistentVolumeClaim
apiVersion:         v1
metadata:
  name:             mypvc
spec:
```

```
  storageClassName: manual
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage:      200Mi
```

接下来，启动该 PVC，并确认它的状态：

```
$ kubectl create -f pvc.yaml
persistentvolumeclaim "mypvc" created

$ kubectl get pv
NAME        CAPACITY  ACCESSMODES  ...  STATUS  CLAIM          STORAGECLASS
hostpathpv  1Gi       RWO          ...  Bound   default/mypvc  manual
```

请注意 PV `hostpathpv` 的状态已经从 `Available`（空闲）变成了 Bound（已绑定）。

最后，在容器内通过 PV 访问数据，这一次我们通过一个部署在文件系统中挂载它。为此，需要创建一个名为 *nginx-usingpv.yaml* 的文件，内容如下：

```
kind:                          Deployment
apiVersion:                    extensions/v1beta1
metadata:
  name:                        nginx-with-pv
spec:
  replicas:                    1
  template:
    metadata:
      labels:
        app:                   nginx
    spec:
      containers:
      - name:                  webserver
        image:                 nginx
        ports:
        - containerPort:       80
        volumeMounts:
        - mountPath:           "/usr/share/nginx/html"
          name:                webservercontent
      volumes:
      - name:                  webservercontent
        persistentVolumeClaim:
          claimName:           mypvc
```

然后启动该部署，如下所示：

```
$ kubectl create -f nginx-using-pv.yaml
deployment "nginx-with-pv" created

$ kubectl get pvc
NAME    STATUS  VOLUME      CAPACITY  ACCESSMODES  STORAGECLASS  AGE
mypvc   Bound   hostpathpv  1Gi       RWO          manual        12m
```

正如你所见，该 PVC 使用了之前创建的 PV。

为了验证数据确实可以保存下来，现在可以创建一个服务（请参阅 5.1 节）和一个 ingress（请参阅 5.5 节），然后用下述命令访问该服务：

```
$ curl -k -s https://192.168.99.100/web
I am content served from a delicious persistent volume
```

很好！你以管理员的身份规定了一个 PV，以开发人员的身份通过 PVC 对其进行了声明，并将其挂载到容器的文件系统中，然后通过部署访问了它。

讨论

在解决方案中，我们使用了 hostPath 类型的 PV。但在正式产品环境中，请不要使用这种 PV，你应该请求集群管理员定义一个由 NFS 或 Amazon Elastic Block Store（简称 EBS）支持的网络卷，从而确保在单个节点出现故障时数据可以保存下来。

请记住 PV 是集群范围内的资源，也就是说，它们不属于命名空间。但是，PVC 则属于命名空间。可以在特定的命名空间内用属于命名空间的 PVC 对 PV 进行声明。

请参阅

- Kubernetes 持久卷文档（*https://kubernetes.io/docs/concepts/storage/persistent-volumes/*）。

- 使用持久卷为pod配置存储空间（*https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/*）。

8.5 掌握 Minikube 上数据的持久性

问题

如何使用 Minikube 在 Kubernetes 上部署一个有状态的应用程序？例如，你想要部署一个 MySQL 数据库。

解决方案

在 pod 定义中使用 PVC 对象（请参阅 8.4 节）和数据库的模板。

首先需要申请一定容量的存储空间。如下 *data.yaml* 清单文件请求了 1 GB 的存储。

```
kind:          PersistentVolumeClaim
apiVersion:    v1
metadata:
  name:        data
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

接下来，在 Minikube 上创建此该 PVC，然后立即查看创建 PV 是否符合以上声明：

```
$ kubectl create -f data.yaml

$ kubectl get pvc
NAME  STATUS  VOLUME                                     CAPACITY ...  ...  ...
data  Bound   pvc-da58c85c-e29a-11e7-ac0b-080027fcc0e7  1Gi      ...  ...  ...

$ kubectl get pv
NAME                                       CAPACITY  ...  ...  ...  ...  ...
pvc-da58c85c-e29a-11e7-ac0b-080027fcc0e7   1Gi       ...  ...  ...  ...  ...
```

现在可以在 pod 中使用该声明了。你需要在清单文件的 volumes 一节中定义卷的名称和 PVC 的类型，并引用上述创建的 PVC。在 volumeMounts 字段中，你可以将该卷挂载到容器内的指定路径中。如果是 MySQL，可以挂载到 /var/lib/mysql 目录中：

```
apiVersion:         v1
kind:               Pod
metadata:
  name:             db
spec:
  containers:
  - image:          mysql:5.5
    name:           db
    volumeMounts:
    - mountPath:    /var/lib/mysql
      name:         data
    env:
      - name:       MYSQL_ROOT_PASSWORD
        value:      root
  volumes:
  - name:           data
    persistentVolumeClaim:
      claimName:    data
```

讨论

Minikube 配置了默认的存储类，该存储类定义了一个默认的 PV provisioner。这意味着当 PVC 创建的时候，Kubernetes 将动态地创建一个相应的 PV 来满足该 PVC。

这就是解决方案中介绍的内容。一旦创建了 data 的 PVC，Kubernetes 就会自动地创建一个 PV 以满足该 PVC。如果仔细观察 Minikube 的默认存储类，可以看到如下 provisioner 类型：

```
$ kubectl get storageclass
NAME                 PROVISIONER
standard (default)   k8s.io/minikube-hostpath

$ kubectl get storageclass standard -o yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
...
provisioner: k8s.io/minikube-hostpath
reclaimPolicy: Delete
```

该存储类使用的存储 provisioner 会创建 hostPath 类型的 PV。通过查看根据之前创建的 PVC 自动生成的 PV 清单文件，可以证实这一点：

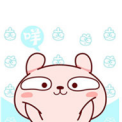
```
$ kubectl get pv
NAME                                       CAPACITY   ... CLAIM           ...
pvc-da58c85c-e29a-11e7-ac0b-080027fcc0e7   1Gi        ... default/foobar  ...

$ kubectl get pv pvc-da58c85c-e29a-11e7-ac0b-080027fcc0e7 -o yaml
apiVersion: v1
kind: PersistentVolume
...
  hostPath:
    path: /tmp/hostpath-provisioner/pvc-da58c85c-e29a-11e7-ac0b-080027fcc0e7
    type: ""
...
```

为了确认刚才创建的卷中保存了数据库的数据文件，你可以连接到 Minikube 并查看目录中的文件：

```
$ minikube ssh
                         _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)

$ ls -l /tmp/hostpath-provisioner/pvc-da58c85c-e29a-11e7-ac0b-080027fcc0e7
total 28688
-rw-rw---- 1 999 999        2 Dec 16 20:02 data.pid
-rw-rw---- 1 999 999  5242880 Dec 16 20:02 ib_logfile0
-rw-rw---- 1 999 999  5242880 Dec 16 20:02 ib_logfile1
-rw-rw---- 1 999 999 18874368 Dec 16 20:02 ibdata1
drwx------ 2 999 999     4096 Dec 16 20:02 mysql
drwx------ 2 999 999     4096 Dec 16 20:03 oreilly
drwx------ 2 999 999     4096 Dec 16 20:02 performance_schema
```

以上信息显示数据确实保存下来了。即使该 pod 宕机（或被删除），数据依然会保存下来。

一般来说，storage classes 允许集群管理员定义各种所需的存储类型。对



于开发人员来说，storage classes 可以抽象存储类型，并方便开发人员直接使用 PVC，而无需担心存储本身。

请参阅

- 有关持久卷的文档（*https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims*）。
- 有关存储类的文档（*https://kubernetes.io/docs/concepts/storage/storage-classes/*）。

8.6 在 GKE 上动态配置持久性存储空间

问题

除了 8.4 节中介绍的通过 PVC 手动配置 PV 之外，如何自动化此过程？也就是说如何根据存储大小或价格需求自动配置 PV？

解决方案

对于 GKE 来说，根据 Saad Ali 的博文“在 Kubernetes 中的动态配置和存储类”（*http://blog.kubernetes.io/2016/10/dynamic-provisioning-and-storage-in-kubernetes.html*）中所介绍的步骤自动完成该操作。

讨论

一般来说，配置与声明 PV 的流程如图 8-2 所示。

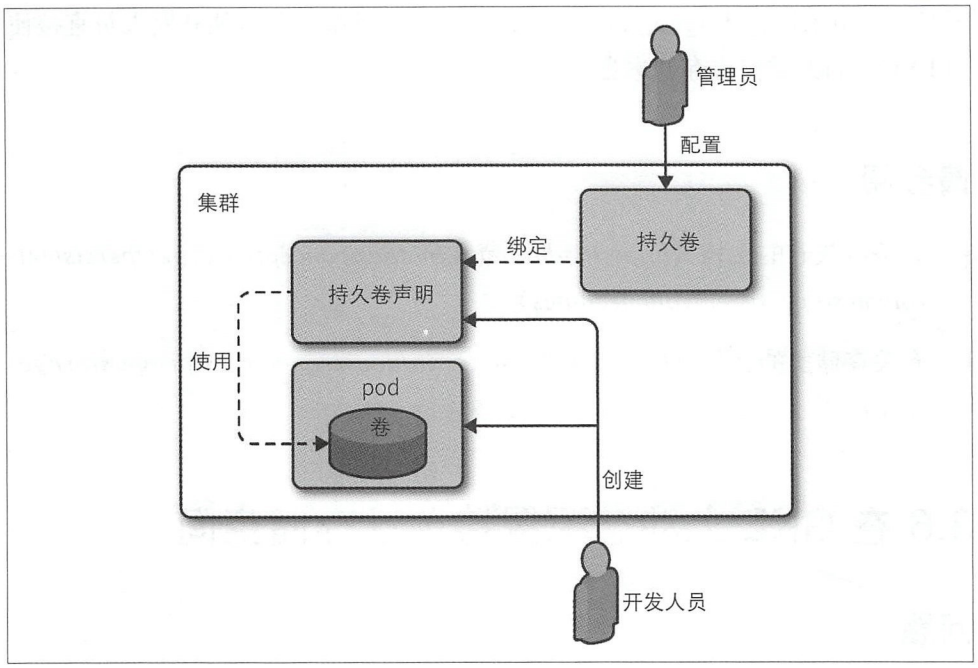
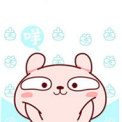


图 8-2：配置与声明 PV 的流程

该流程需要管理员与开发人员协调卷的类型与大小，而且我们可以通过动态配置简化此流程。



第 9 章

伸缩

Kubernetes 中的伸缩对于不同的用户有着不同的意义。我们大致可以将其分为两类：

- 集群的伸缩，有时被称为基础设施级的伸缩，指的是根据集群利用率向集群添加或删除工作节点的（自动化）处理。
- 应用程序级的伸缩，有时也称为 pod 伸缩，指的是根据各项指标调整 pod 特征的（自动化）处理，这些指标包括 CPU 利用率等底层信号，也包括 pod 每秒服务的 HTTP 请求等高级信号。pod 级的伸缩主要包括以下两种：
  - 横向自动伸缩（Horizontal Pod Autoscalers，HPA），指的是根据特定的指标增加或减少 pod 副本的个数。
  - 纵向自动伸缩（Vertical Pod Autoscalers，VPA），指的是增加或减少 pod 内运行的容器的资源需求。截至 2018 年 1 月 VPA 尚在开发中，因此我们不打算在本书中讨论 VPA。如果你对这个话题感兴趣，可以参阅 Michael 的博文“不容忽视的容器资源消耗”（*https://hackernoon.com/container-resource-consumption-too-important-to-ignore-7484609a3bb7*）。

在本章中，我们首先介绍 AWS 和 GKE 的集群伸缩，然后讨论应用程序级别的横向自动伸缩。

9.1 部署的伸缩

问题

如何横向伸缩已有的部署？

解决方案

可以使用 `kubectl scale` 命令对部署进行伸缩操作。

这里以 4.4 节中介绍的 `fancyap` 部署为例，它含有 5 个副本。如果环境中没有该部署，那么可以通过运行命令 `kubectl create -f fancyapp.yaml` 来创建。

现在假设该部署上的负荷降低了，不再需要 5 个副本，3 个就够了。那么可以通过下列步骤，将该部署收缩到 3 个副本：

```
$ kubectl get deploy fancyapp
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
fancyapp   5         5         5            5           9m

$ kubectl scale deployment fancyapp --replicas=3
deployment "fancyapp" scaled

$ kubectl get deploy fancyapp
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
fancyapp   3         3         3            3           10m
```

如果不想手动伸缩部署，也可以自动化该处理，请参阅 9.4 节中介绍的例子。

9.2 在 GKE 中自动调整集群的大小

问题

如何根据集群的利用率，自动扩展或缩减集群上的节点数目？

解决方案

可以使用 GKE 的集群自动伸缩。本节我们假设你已经安装了 `gcloud` 命令，并且设置好了环境（即创建了项目并设置好了付款方式）。

首先，创建拥有一个工作节点的集群，并确保可以通过 kubectl 访问：

```
$ gcloud container clusters create --num-nodes=1 supersizeme
Creating cluster supersizeme...done.
Created [https://container.googleapis.com/v1/projects/k8s-cookbook/zones/...].
kubeconfig entry generated for supersizeme.
NAME         ZONE            MASTER_VERSION  MASTER_IP       ...  STATUS
supersizeme  europe-west2-b  1.7.8-gke.0     35.189.116.207  ...  RUNNING

$ gcloud container clusters get-credentials supersizeme --zone europe-west2-b \
                                                        --project k8s-cookbook
```

接下来，启用集群的自动伸缩：

```
$ gcloud beta container clusters update supersizeme --enable-autoscaling \
                                                    --min-nodes=1 --max-nodes=3 \
                                                    --zone=europe-west2-b \
                                                    --node-pool=default-pool
```

请注意如果 `beta` 命令群没有启用，那么这步中会提示你安装该命令。

这时如果查看 Google Cloud 的控制台，可以看到图 9-1 所示的内容。

现在通过部署启动 15 个 pod。这可以生成足够的负荷来触发该集群的自动伸缩：

```
$ kubectl run ghost --image=ghost:0.9 --replicas=15
```

现在该集群应该拥有 3 个节点，如图 9-2 所示。

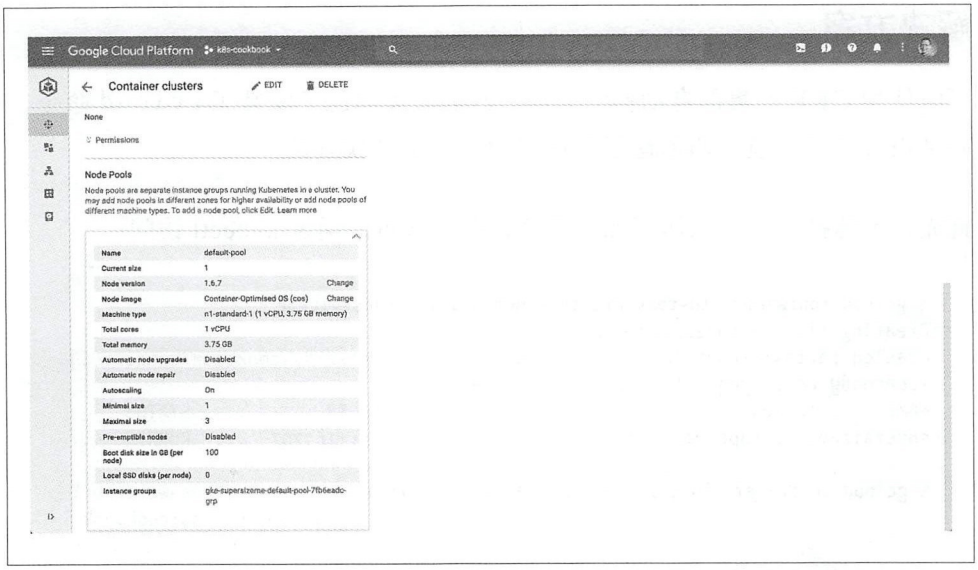


图 9-1：Google Cloud 控制台的截图，显示了初始的集群只有一个节点

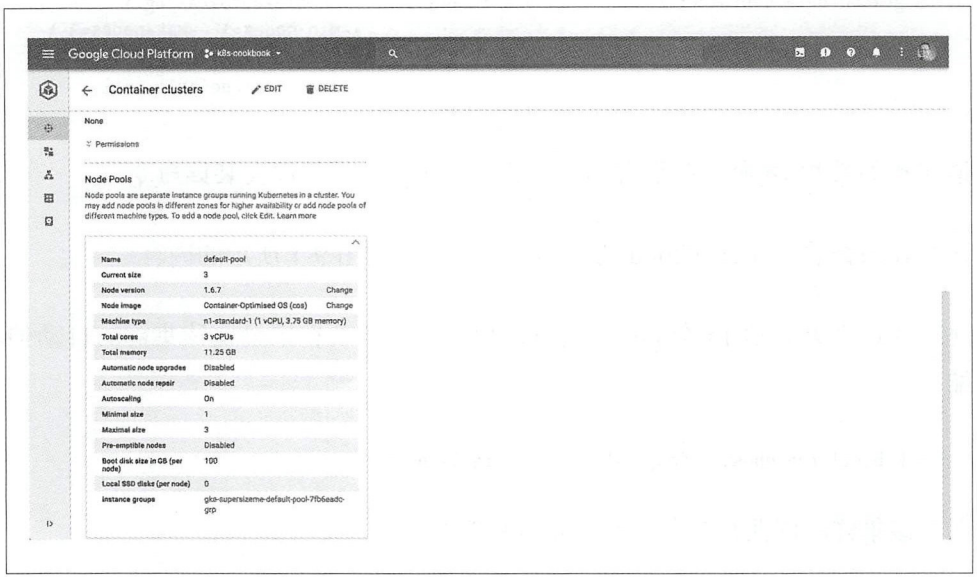
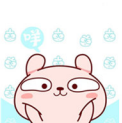


图 9-2：Google Cloud 控制台的截图，显示了现在集群拥有 3 个节点

图 9-3 展示了整个交互的过程：



- 在左上的终端窗口，可以看到负载（创建了 15 个 pod，触发了该集群的自动伸缩事件）。
- 在右上的终端窗口，可以看到 gcloud 命令启用了该集群的自动伸缩。
- 在下面的终端窗口，可以看到 kubectl get nodes --watch 命令的返回结果，显示了当前可用的节点。

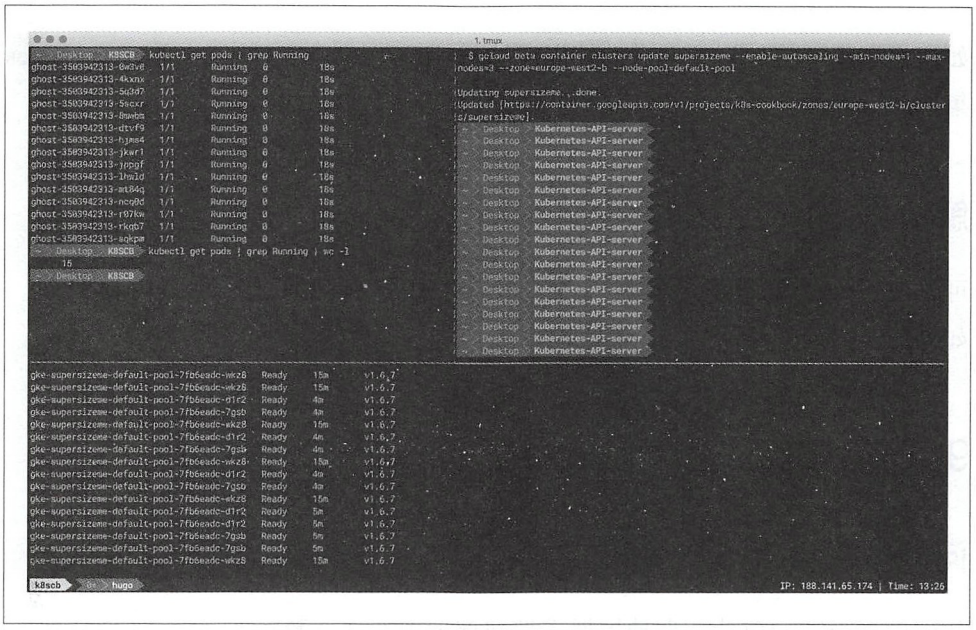


图 9-3：各终端窗口的截图，显示了集群正在进行自动伸缩

请注意节点池中所有的节点都应该有相同的容量、标签，每个节点上运行的系统 pod 也应该是相同的。为节点池指定最大容量时，请检查配额是否足够大。

在使用结束后，请不要忘了执行 gcloud container clusters delete supersizeme；否则你需要继续为集群资源付费。

请参阅

- *kubernetes/autoscaler* 代码仓库中的集群自动伸缩（*https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler*）。

- GKE 文档中关于集群的自动伸缩（*https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler*）。

9.3 在 AWS 中自动调整集群的大小

问题

如何根据集群的利用率，自动扩展或缩减运行在 AWS EC2 之上的 Kubernetes 集群的节点数目？

解决方案

可以使用 AWS 的集群自动伸缩，它会利用 AWS 自动伸缩组的 Helm 软件包。如果还未安装 Helm，请先参阅 14.1 节。

9.4 在 GKE 上使用 pod 的横向自动伸缩

问题

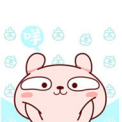
如何根据当前的负荷，自动扩展或缩减部署中的 pod 数目？

解决方案

可以使用横向自动伸缩（Horizontal Pod Autoscalers，HPA），如下所述。

首先，创建一个应用作为 HPA 的对象，例如一个 PHP 环境和服务器：

```
$ kubectl run appserver --image=gcr.io/google_containers/hpa-example \
                        --requests=cpu=200m --expose --port=80
service "appserver" created
   | NAME    ZONE            MASTER_VERSION
deployment "appserver" created
```

接下来，创建 HPA，并定义触发参数 `--cpu-percent=40`，意思是说 CPU 的利用率不能高于 40%：

```
$ kubectl autoscale deployment appserver --cpu-percent=40 --min=1 --max=5
deployment "appserver" autoscaled

$ kubectl get hpa --watch
NAME        REFERENCE              TARGETS           MINPODS   MAXPODS   REPLICAS   AGE
appserver   Deployment/appserver   <unknown> / 40%   1         5         0          14s
```

打开第二个终端窗口，注意观察该部署：

```
$ kubectl get deploy appserver --watch
```

最后，打开第三个终端窗口，启动负荷生成器：

```
$ kubectl run -i -t loadgen --image=busybox /bin/sh
If you don't see a command prompt, try pressing enter.

/ # while true; do wget -q -O- http://appserver.default.svc.cluster.local; done
```

此处我们同时用到了 3 个终端窗口，整体状况如图 9-4 所示。

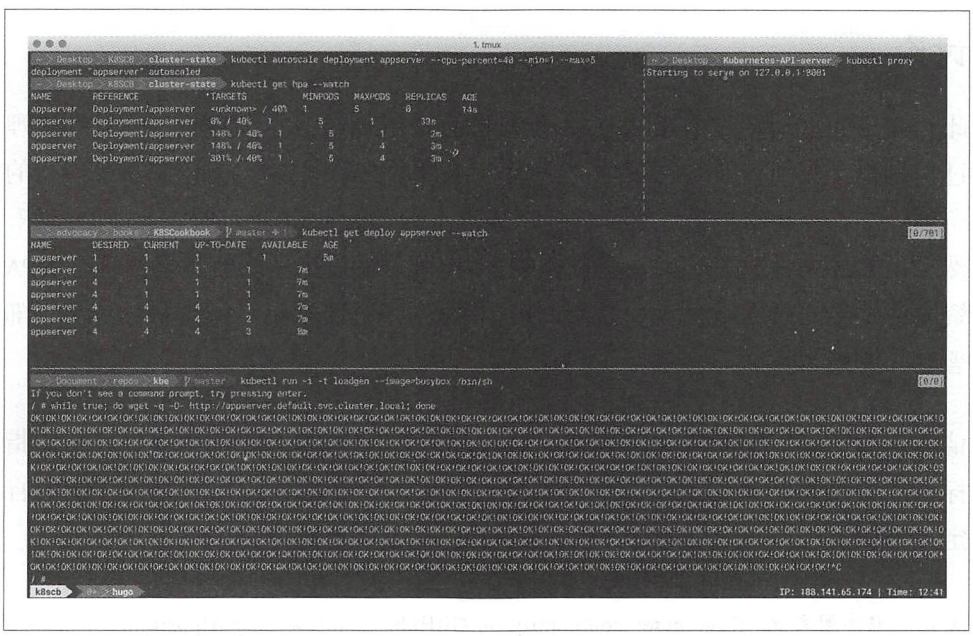


图 9-4：设置 HPA 时的终端窗口的截图

这次我们可以在Kubernetes仪表盘上看到该appserver部署上的HPA的效果，如图 9-5 所示。

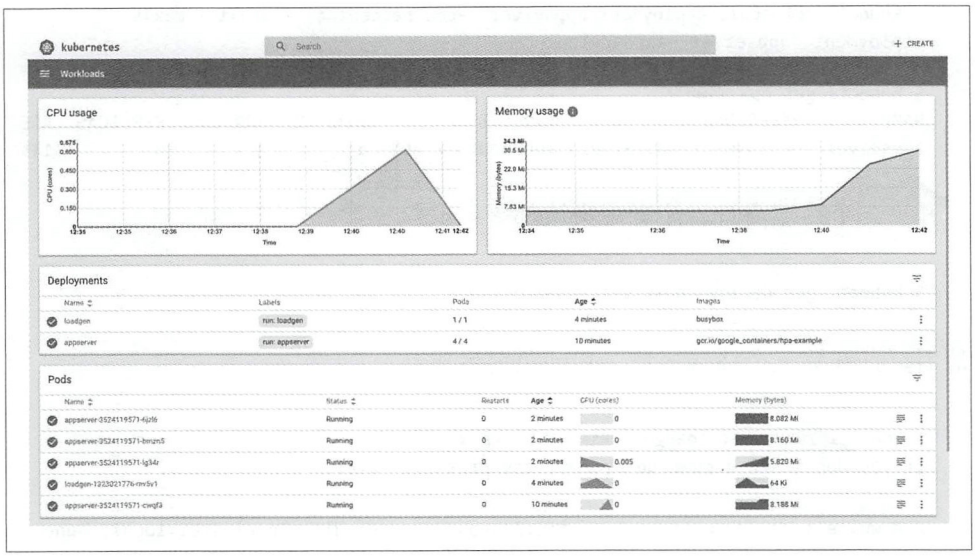


图 9-5：Kubernete 仪表盘的截图，显示了 HPA 的效果

讨论

此处介绍的自动伸缩是通过 HPA 控制器自动增加或减少副本数目的自动处理过程，其会受到 HPA 资源的影响。作为 Kubernetes 控制层内控制器管理器的一部分，该控制器将通过集群中各个节点上运行的 cAdvisor 检查 pod 的指标，然后由 Heapster 进行汇总。再交由 HPA 控制器计算副本的数目是否符合 HPA 资源中定义的目标指数[注 1]。最后 HPA 根据这个计算结果，调整目标资源（部署等）的副本数。

请注意自动伸缩的配置比较复杂，而且调节 CPU 或 RAM 利用率等底层的指标可能会带来意想不到的影响。如果可以的话，请尽量使用应用程序级的自定义指标（*https://blog.openshift.com/kubernetes-1-8-now-custom-metrics/*）。

注 1：请参阅文档：Kubernetes community on GitHub, “Autoscaling Algorithm”（*https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/horizontal-pod-autoscaler.md#autoscaling-algorithm*）。

请参阅

- HPA 的操作实例（*https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/*）。
- Jerzy Szczepkowski 和 Marcin Wielgus 的博文“Kubernetes 的自动伸缩”（*http://blog.kubernetes.io/2016/07/autoscaling-in-kubernetes.html*）。
- GKE 中自动伸缩的演示（*https://github.com/mhausenblas/k8s-autoscale*）。

第 10 章

安全

在 Kubernetes 上运行应用程序需要开发人员和运营人员共同承担责任，将恶意攻击降到最小，贯彻最小特权的原则，并清晰地定义对资源的访问权限。在本章中，我们将通过各节介绍用户可以和应该执行的原则，从而确保集群和应用安全地运行。本章的各节所涵盖的内容包括：

- 服务账号的角色和使用。
- 基于角色的访问控制（Role-Based Access Control，RBAC）。
- 定义 pod 的安全上下文环境。

10.1 赋予应用程序唯一的身份

问题

如何赋予应用程序唯一的身份，以便从细节上控制访问的资源？

解决方案

创建一个服务账号，并在 pod 规格中使用这个账号。

首先，创建一个名为 myappsa 的新服务账号，并仔细观察它：

```
$ kubectl create serviceaccount myappsa
serviceaccount "myappsa" created
```

```
$ kubectl describe sa myappsa
Name:           myappsa
Namespace:      default

Labels:         <none>
Annotations:    <none>

Image pull secrets:     <none>

Mountable secrets:      myappsa-token-rr6jc

Tokens:                 myappsa-token-rr6jc

$ kubectl describe secret myappsa-token-rr6jc
Name:           myappsa-token-rr6jc
Namespace:      default
Labels:         <none>
Annotations:    kubernetes.io/service-account.name=myappsa
                kubernetes.io/service-account.uid=0baa3df5-c474-11e7-8f08...

Type:   kubernetes.io/service-account-token

Data
====
ca.crt:         1066 bytes
namespace:      7 bytes
token:          eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9 ...
```

接下来在 pod 中使用这个服务，如下所示：

```
kind:                Pod
apiVersion:          v1
metadata:
  name:              myapp
spec:
  serviceAccountName: myappsa
  containers:
  - name:            main
    image:           centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
```

然后通过运行来确认 pod 正确使用了该服务账号 myappsa：

```
$ kubectl exec myapp -c main \
         cat  /var/run/secrets/kubernetes.io/serviceaccount/token \
         eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9 ...
```

可以看到，myappsa 服务账号的 token 确实挂载到了 pod 中的预期位置，接下来可以使用该服务账号了。

服务账号本身不是特别有用，但是它可以让我们从细节上控制访问，更多信息请参阅 10.2 节。

讨论

可以识别的实体是认证和授权的前提条件。从 API 服务器的角度看来，有两种类型的实体：终端用户（人类）和应用程序。尽管用户身份（的管理）不属于 Kubernetes 的范围，但是它提供了代表应用身份的一级资源：服务账号。

严格来说，应用程序的认证由 token 负责，并保存在本地 `/var/run/secrets/kubernetes.io/serviceaccount/token` 的文件中，该文件会通过 secret 自动挂载。服务账号属于命名空间的资源，访问形式如下所示：

```
system:serviceaccount:$NAMESPACE:$SERVICEACCOUNT
```

试着查看某个命名空间中的服务账号，可以看到类似以下的信息：

```
$ kubectl get sa
NAME         SECRETS   AGE
default      1         90d
myappsa      1         19m
prometheus   1         89d
```

请注意这里有个名为 `default` 的服务账号，它是自动创建的。如果不像上述解决方案中那样，明确为 pod 指定服务账号，那么系统会自动在该 pod 的命名空间内为其分配 `default` 服务账号。

请参阅

- 管理服务账号（*https://kubernetes.io/docs/admin/service-accounts-admin/*）。

- 为 Pod 配置服务账号（*https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/*）。
- 从私有注册表中提取映像（*https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/*）。

10.2 列举并查看访问控制信息

问题

如何查看有哪些操作可以使用？例如，如何更新部署或列举 secret？

解决方案

下述的解决方案假设你采用的授权方式是基于角色的访问控制（Role-Based Access Control，RBAC）。

可以通过 `kubectl auth can-i` 的命令，查看某个用户是否允许使用针对指定资源的某项操作。例如，可以执行该命令来检查服务账号 `system:serviceaccount:sec:myappsa` 是否可以查看命名空间 `sec` 中的 pod 列表：

```
$ kubectl auth can-i list pods --as=system:serviceaccount:sec:myappsa -n=sec
yes
```

如果你想在 Minikube 中尝试该操作，那么需要在执行安装文件的时候，加入参数：`--extra-config=apiserver.Authorization.Mode=RBAC`。

可以通过如下命令查看命名空间中的角色列表：

```
$ kubectl get roles -n=kube-system
NAME                                             AGE
extension-apiserver-authentication-reader        1d
```

```
system::leader-locking-kube-controller-manager   1d
system::leader-locking-kube-scheduler            1d
system:controller:bootstrap-signer               1d
system:controller:cloud-provider                 1d
system:controller:token-cleaner                  1d

$ kubectl get clusterroles -n=kube-system
NAME                                             AGE
admin                                            1d
cluster-admin                                    1d
edit                                             1d
system:auth-delegator                            1d
system:basic-user                                1d
system:controller:attachdetach-controller        1d
system:controller:certificate-controller         1d
system:controller:cronjob-controller             1d
system:controller:daemon-set-controller          1d
system:controller:deployment-controller          1d
system:controller:disruption-controller          1d
system:controller:endpoint-controller            1d
system:controller:generic-garbage-collector      1d
system:controller:horizontal-pod-autoscaler      1d
system:controller:job-controller                 1d
system:controller:namespace-controller           1d
system:controller:node-controller                1d
system:controller:persistent-volume-binder       1d
system:controller:pod-garbage-collector          1d
system:controller:replicaset-controller          1d
system:controller:replication-controller         1d
system:controller:resourcequota-controller       1d
system:controller:route-controller               1d
system:controller:service-account-controller     1d
system:controller:service-controller             1d
system:controller:statefulset-controller         1d
system:controller:ttl-controller                 1d
system:discovery                                 1d
system:heapster                                  1d
system:kube-aggregator                           1d
system:kube-controller-manager                   1d
system:kube-dns                                  1d
system:kube-scheduler                            1d
system:node                                      1d
system:node-bootstrapper                         1d
system:node-problem-detector                     1d
system:node-proxier                              1d
system:persistent-volume-provisioner             1d
view                                             1d
```

上述输出结果显示了预定义的角色，这些角色可以直接用于用户账号和服务账号。

为了进一步查看某个角色，并了解其允许的操作，可以使用：

```
$ kubectl describe clusterroles/view -n=kube-system
Name:           view
Labels:         kubernetes.io/bootstrapping=rbac-defaults
Annotations:    rbac.authorization.kubernetes.io/autoupdate=true
PolicyRule:
  Resources                                  Non-Resource URLs     ...  ...
  ---------                                  -----------------     ---  ---
  bindings                                   []                    ...  ...
  configmaps                                 []                    ...  ...
  cronjobs.batch                             []                    ...  ...
  daemonsets.extensions                      []                    ...  ...
  deployments.apps                           []                    ...  ...
  deployments.extensions                     []                    ...  ...
  deployments.apps/scale                     []                    ...  ...
  deployments.extensions/scale               []                    ...  ...
  endpoints                                  []                    ...  ...
  events                                     []                    ...  ...
  horizontalpodautoscalers.autoscaling       []                    ...  ...
  ingresses.extensions                       []                    ...  ...
  jobs.batch                                 []                    ...  ...
  limitranges                                []                    ...  ...
  namespaces                                 []                    ...  ...
  namespaces/status                          []                    ...  ...
  persistentvolumeclaims                     []                    ...  ...
  pods                                       []                    ...  ...
  pods/log                                   []                    ...  ...
  pods/status                                []                    ...  ...
  replicasets.extensions                     []                    ...  ...
  replicasets.extensions/scale               []                    ...  ...
  replicationcontrollers                     []                    ...  ...
  replicationcontrollers/scale               []                    ...  ...
  replicationcontrollers.extensions/scale    []                    ...  ...
  replicationcontrollers/status              []                    ...  ...
  resourcequotas                             []                    ...  ...
  resourcequotas/status                      []                    ...  ...
  scheduledjobs.batch                        []                    ...  ...
  serviceaccounts                            []                    ...  ...
  services                                   []                    ...  ...
  statefulsets.apps                          []                    ...  ...
```

除了 kube-system 命名空间中定义的默认角色外，你还可以自定义角色，请参阅 10.3 节。

如果RBAC被激活，在很多环境（包括Minikube和GKE）中，当访问Kubernetes的仪表盘的时候，可能会看到Forbidden（403）的状态码和如下错误信息：

> User “system:serviceaccount:kube-system:default” cannot list pods in the namespace “sec”. (get pods)

如果想要访问仪表盘，必须赋予kubesystem:default服务账号必要的权限：

```
$ kubectl create clusterrolebinding admin4kubesystem \
  --clusterrole=cluster-admin \
  --serviceaccount=kube-system:default
```

请注意这个命令会将很多权限都赋予这个服务账号，因此不建议在生产环境中这么做。

讨论

如图10-1所示，RBAC的授权方式有以下几部分组成：

- 实体：指一个组、用户或服务账号。
- 资源：例如pod、服务或加密信息。
- 角色：定义了访问资源的行为规则。
- 角色绑定：将角色赋予实体。

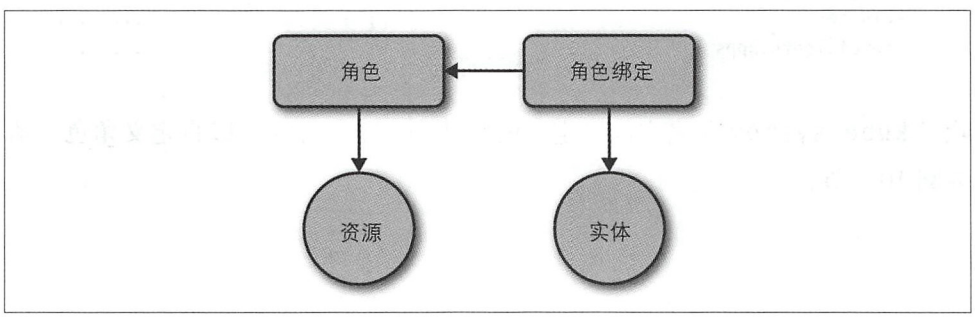


图10-1：RBAC概念图

一个角色在规则内可以对资源进行的操作包括：

- `get、list、watch`
- `create`
- `update/patch`
- `delete`

我们可以将角色分成如下两种类型：

- 集群范围的角色：集群角色和它们相应的集群角色绑定。
- 命名空间范围的角色：角色和角色绑定。

在 10.3 节中，我们将深入讨论如何创建用户自定义规则，并应用到角色和资源中。

请参阅

- Kubernetes 授权方式综述（*https://kubernetes.io/docs/admin/authorization/*）。
- 使用 RBAC 的授权方式（*https://kubernetes.io/docs/admin/authorization/rbac/*）。

10.3 控制资源的访问权限

问题

对于既定的用户或应用程序，如何允许或拒绝其特定操作，例如查看 secret 或更新部署等？

解决方案

假设你想要限制一个应用，只允许它查看 pod，即查看 pod 列表和查看 pod 的详细信息。

首先使用专门的服务账号 `myappsa`（请参阅 10.1 节），在 YAML 清单文件 *pod-with-sa.yaml* 中定义 pod：

```
kind:                Pod
apiVersion:          v1
metadata:
  name:              myapp
  namespace:         sec
spec:
  serviceAccountName: myappsa
  containers:
  - name:            main
    image:           centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
```

接下来，在 *pod-reader.yaml* 清单文件中定义一个名为 `podreader` 的角色，该清单文件定义了允许的资源访问操作：

```
kind:       Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name:      podreader
  namespace: sec
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs:     ["get", "list"]
```

最后，别忘了需要通过 *pod-reader-binding.yaml* 文件中定义的角色绑定，将这个角色 `podreader` 赋予服务账号 `myappsa`：

```
kind:       RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name:      podreaderbinding
  namespace: sec
```

```
roleRef:
  apiGroup:  rbac.authorization.k8s.io
  kind:      Role
  name:      podreader
subjects:
- kind:      ServiceAccount
  name:      myappsa
  namespace: sec
```

在创建各项资源的时候，可以直接使用 YAML 清单文件（假设服务账号已经创建完毕）：

```
$ kubectl create -f pod-reader.yaml
$ kubectl create -f pod-reader-binding.yaml
$ kubectl create -f pod-with-sa.yaml
```

除了分别为角色和角色绑定创建清单文件外，你还可以使用如下命令：

```
$ kubectl create role podreader \
          --verb=get --verb=list \
          --resource=pods -n=sec

$ kubectl create rolebinding podreaderbinding \
          --role=sec:podreader \
          --serviceaccount=sec:myappsa \
          --namespace=sec -n=sec
```

请注意这个例子是命名空间内的访问控制设置，因为上述使用的是角色和角色的绑定。对于集群范围的访问控制，需要使用相应的 `create clusterrole` 和 `create clusterrolebinding` 命令。

有时不易判断应该使用角色还是集群角色以及角色绑定，所以在这里介绍几个实用的经验规则：

- 如果想在特定命名空间中，限制对命名空间资源（如服务或 pod）的访问，那么可以使用角色和角色绑定（请参考本节的做法）。
- 如果想在多个命名空间中重复使用一个角色，那么可以使用集群角色和角色绑定。
- 如果想限制对节点等集群范围资源的访问，或限制所有命名空间都有的命名空间资源的访问，那么可以使用集群角色和角色绑定。

请参阅

- 在 Kubernetes 集群内配置 RBAC（*https://docs.bitnami.com/kubernetes/how-to/configure-rbac-in-your-kubernetes-cluster/*）。
- Antoine Cotten 的博文“Kubernetes v1.7 安全实战经验”（*https://acotten.com/post/kube17-security*）。

10.4 加强 pod 的安全

问题

如何在 pod 级别为应用定义安全上下文环境？例如，以非特权进程运行该应用，或限制该应用可以访问的卷的类别。

解决方案

可以使用 pod 规格中的 `securityContext` 字段，加强 Kubernetes 中 pod 级别的安全策略。

假设你想用非 root 用户运行一个应用，那么需要使用容器级别的安全上下文环境，如下 *securedpod.yaml* 清单文件所示：

```
kind:                Pod
apiVersion:          v1
metadata:
  name:              secpod
spec:
  containers:
  - name:            shell
    image:           centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    securityContext:
      runAsUser:     5000
```

现在创建该 pod，并检查运行该容器的用户：

```
$ kubectl create -f securedpod.yaml
pod "secpod" created

$ kubectl exec secpod ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
5000         1  0.0  0.0   4328   672 ?        Ss   12:39   0:00 sleep 10000
5000         8  0.0  0.1  47460  3108 ?        Rs   12:40   0:00 ps aux
```

不出所料，该 pod 是由用户 ID：5000 运行的。请注意除了具体的容器之外，现在还可以在 pod 上使用 `securityContext`。

加强 pod 级别安全策略的另一个更强大的方法是使用 pod 的安全策略（pod security policies，PSP）。这是集群范围的资源，可以通过它们定义一系列的策略，包括一些与上述类似的操作，以及与存储和网络相关的限制。关于 PSP 的使用指南，请参阅 Bitnami 中的 Kubernetes 文档“通过 pod 安全策略保护 Kubernetes 集群”（*https://docs.bitnami.com/kubernetes/how-to/secure-kubernetes-cluster-psp/*）。

请参阅

- Pod 安全策略文档（*https://kubernetes.io/docs/concepts/policy/pod-security-policy/*）。
- pod 或容器的安全环境配置（*https://kubernetes.io/docs/tasks/configure-pod-container/security-context/*）。

第 11 章

监控与日志

在本章中，我们将集中介绍基础设施与应用程序层的监控与日志。在 Kubernetes 的环境中，不同的角色通常有不同的职责：

- 管理员角色：如集群管理员、网络运营人员、或命名空间管理员等关注基础设施方面的工作人员。常见的问题可能有：节点是否健康？我们是否应该增设一个工作节点？集群的利用率是多少？用户的配额是否快用完了？
- 开发人员角色：主要考虑和操作他们的应用程序的上下文环境，在目前的微服务时代，他们可能要关心几个甚至十几个应用程序。例如，承担开发角色的工作人员可能会问：我是否为我的应用分配了足够的资源？我的应用应该扩展到多少个副本？我是否可以访问正确的卷，以及还有多少容量？是否有运行失败的应用，如果有，原因是什么？

我们首先利用 Kubernetes 的存活探针和就绪探针，来讨论集群内部的监视，然后集中讨论如何利用 Heapster 和 Prometheus 来监视集群，最后我们将介绍日志相关的内容。

11.1 访问容器的日志

问题

如何访问容器（该容器在指定的 pod 内运行）内运行的应用程序的日志？

解决方案

可以使用 kubectl logs 命令。通过以下命令可以查看该命令的各项参数以及用法：

```
$ kubectl logs --help | more
Print the logs for a container in a pod or specified resource. If the pod has only
one container, the container name is optional.

Aliases:
logs, log

Examples:
  # Return snapshot logs from pod nginx with only one container
  kubectl logs nginx
...
```

例如，对于一个由部署启动的 pod（请参阅 4.1 节），可以通过如下命令查看日志：

```
$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
ghost-8449997474-kn86m        1/1       Running   0          1m

$ kubectl logs ghost-8449997474-kn86m
[2017-12-16 18:44:18] INFO Creating table: posts
[2017-12-16 18:44:18] INFO Creating table: users
[2017-12-16 18:44:18] INFO Creating table: roles
[2017-12-16 18:44:18] INFO Creating table: roles_users
...
```

如果 pod 有多个容器，那么可以使用 kubectl logs 的 -c 参数指定容器的名字，从而获取它们的日志。

11.2 使用存活探针修复失败状态

问题

如何确保当 pod 中运行的应用程序进入失败状态时，Kubernetes 会自动重启 pod？

解决方案

可以使用存活探针[注1]。如果探针失败，那么 kubelet 会自动重启 pod。探针是 pod 规格的一部分，可以添加到 containers 一节的字段中。pod 中的每个容器都可以拥有一个存活探针。

探针可以有三种类型：它可以是容器内部运行的命令；也可以是一个 HTTP 请求，指向容器内由网络服务提供的特定路径；或者是更通用的 TCP 探针。

下面的例子展示了一个基本的 HTTP 探针：

```
apiVersion:   v1
kind:         Pod
metadata:
  name:       liveness-nginx
spec:
  containers:
  - name:     liveness
    image:    nginx
    livenessProbe:
      httpGet:
        path: /
        port: 80
```

完整的例子请参阅 11.4 节。

请参阅

- Kubernetes 容器探针的相关文档（*https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes*）。

注 1：请参阅文档：Kubernetes, “Configure Liveness and Readiness Probes”（*https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/#define-a-liveness-command*）。

11.3 使用就绪探针来控制 pod 的访问流

问题

存活探针（请参阅 11.2 节）能够指示 pod 启动和运行的状态，但是如何确保在应用程序准备好服务请求之后再接受访问？

解决方案

可以向 pod 规格添加就绪探针[注2]。与存活探针类似，就绪探针也有三种类型（具体信息请参阅相关文档）。下面是一个简单的例子，其中就绪探针在 `nginx` Docker 映像的单个 pod 上运行。就绪探针向端口 80 发送了一个 HTTP 的请求：

```
apiVersion:   v1
kind:         Pod
metadata:
  name:       readiness-nginx
spec:
  containers:
  - name:     readiness
    image:    nginx
    readinessProbe:
      httpGet:
        path:  /
        port:  80
```

讨论

虽然上面介绍的就绪探针与 11.2 节中所介绍的存活探针相同，但一般情况下两者是不一样的，因为它们旨在提供应用程序不同方面的信息。存活探针负责查看应用程序进程是否处于活动状态，但是应用程序可能并没有准备好接

注 2：请参阅文档：Kubernetes, "Define readiness probes"（*https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/#define-readiness-probes*）。

收请求。而就绪探针负责检查应用程序是否可以正确服务请求。因此，只有当通过就绪探针的检查，pod 才能成为服务（请参阅 5.1 节）的一部分。

请参阅

- Kubernetes 容器探针的相关文档（*https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes*）。

11.4 向部署添加存活探针和就绪探针

问题

如何自动检查应用是否健康，且在不健康的时候让 Kubernetes 做相应的处理？

解决方案

为了通知 Kubernetes 应用的状况，可以添加存活探针和就绪探针，如下所示。

首先定义一个部署清单文件 *webserver.yaml*：

```
apiVersion:              extensions/v1beta1
kind:                    Deployment
metadata:
  name:                  webserver
spec:
  replicas:              1
  template:
    metadata:
      labels:
        app:             nginx
    spec:
      containers:
      - name:            nginx
        image:           nginx:stable
        ports:
        - containerPort: 80
```

在 pod 规格中的 containers 一节中定义存活探针和就绪探针。请参阅上述介绍的例子（请参阅 11.2 节和 11.3 节），并向部署 pod 模板的容器规格中添加如下内容：

```
...
    livenessProbe:
      initialDelaySecond: 2
      periodSeconds: 10
      httpGet:
        path: /
        port: 80
    readinessProbe:
      initialDelaySecond: 2
      periodSeconds: 10
      httpGet:
        path: /
        port: 80
...
```

现在启动部署并检查探针：

```
$ kubectl create -f webserver.yaml

$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
webserver-4288715076-dk9c7    1/1       Running   0          2m

$ kubectl describe pod/webserver-4288715076-dk9c7
Name:           webserver-4288715076-dk9c7
Namespace:      default
Node:           node/172.17.0.128
...
Status:         Running
IP:             10.32.0.2
Controllers:    ReplicaSet/webserver-4288715076
Containers:
  nginx:
    ...
    Ready:          True
    Restart Count:  0
    Liveness:       http-get http://:80/ delay=2s timeout=1s period=10s #...
...
```

请注意 kubectl describe 的返回结果实际上还有大段的信息，但是与我们此处的问题不相干，所以在这里做了省略，只显示了重要内容。

讨论

为了确认 pod 内的容器是否健康、是否可以接受访问，Kubernetes 提供了一系列的健康检查机制。在 Kubernetes 里，健康检查称为探针，定义在容器一级，而非 pod 一级，它由两个不同的组成部分：

- 每个工作节点上的 kubelet 使用规格中的 livenessProbe 指令决定什么时候重启容器。这些存活探针可以帮助应付突发问题或死锁。
- 一套 pod 的服务负载平衡使用 `readinessProbe` 指令决定是否 pod 准备好并可以接受访问了。如果没有准备好，就从服务器的访问点池中将该 pod 排除在外。请注意只有当所有容器都准备就绪，pod 才会被当成准备就绪。

至于何时该选用哪种探针，应当根据容器的行为进行选择。如果在探测失败的时候，容器可以并且应该被杀掉重启，那么请使用存活探针，并将 `restartPolicy` 设置为 `Always` 或 `OnFailure`。如果想在 pod 准备就绪之后再接受访问，那么可以使用就绪探针。请注意后者的情况下，就绪探针也起到了存活探针的作用。

请参阅

- 配置存活探针与就绪探针（*https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/*）。
- Pod 生命周期的相关文档（*https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/*）。
- 初始化容器的相关文档（在 v1.6 及之后的版本中相对稳定）（*https://kubernetes.io/docs/concepts/workloads/pods/init-containers/*）。

11.5 在 Minikube 上激活 Heapster 监视资源

问题

如果需要在 Minikube 中使用 kubectl top 命令监视资源使用状况的时候，但遇到了 Heapster 插件未运行的错误，如何解决？

```
$ kubectl top pods
Error from server (NotFound): the server could not find the requested resource
(get services http:heapster:)
```

解决方案

最新版本的 minikube 命令包含了插件管理器，可以在此管理器上通过一个命令激活 Heapster 以及其他插件，如 ingress 控制器等：

```
$ minikube addons enable heapster
```

启用 Heapster 插件会在 kube-system 命名空间中生成两个 pod：一个 pod 运行 Heapster，另外一个运行 InfluxDB 时间序列数据库和 Grafana 的仪表盘。

等待几分钟，在收集到第一个指标后，kubectl top 命令即可正常返回资源的指标：

```
$ kubectl top node
NAME       CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
minikube   187m         9%        1154Mi          60%

$ kubectl top pods --all-namespaces
NAMESPACE     NAME                          CPU(cores)   MEMORY(bytes)
default       ghost-2663835528-fb044        0m           140Mi
```

```
kube-system   kube-dns-v20-4bkhn                 3m          12Mi
kube-system   heapster-6j5m8                     0m          21Mi
kube-system   influxdb-grafana-vw9x1             23m         37Mi
kube-system   kube-addon-manager-minikube        47m         3Mi
kube-system   kubernetes-dashboard-scsnx         0m          14Mi
kube-system   default-http-backend-75m71         0m          1Mi
kube-system   nginx-ingress-controller-p8fmd     4m          51Mi
```

现在可以访问 Grafana 的仪表盘，并根据喜好自定义显示的内容：

```
$ minikube service monitoring-grafana -n kube-system
Waiting, endpoint for service is not ready yet...
Waiting, endpoint for service is not ready yet...
Waiting, endpoint for service is not ready yet...
Waiting, endpoint for service is not ready yet...
Opening kubernetes service kube-system/monitoring-grafana in default browser...
```

该命令运行完毕后，浏览器将自动打开，并显示如图 11-1 所示的内容。

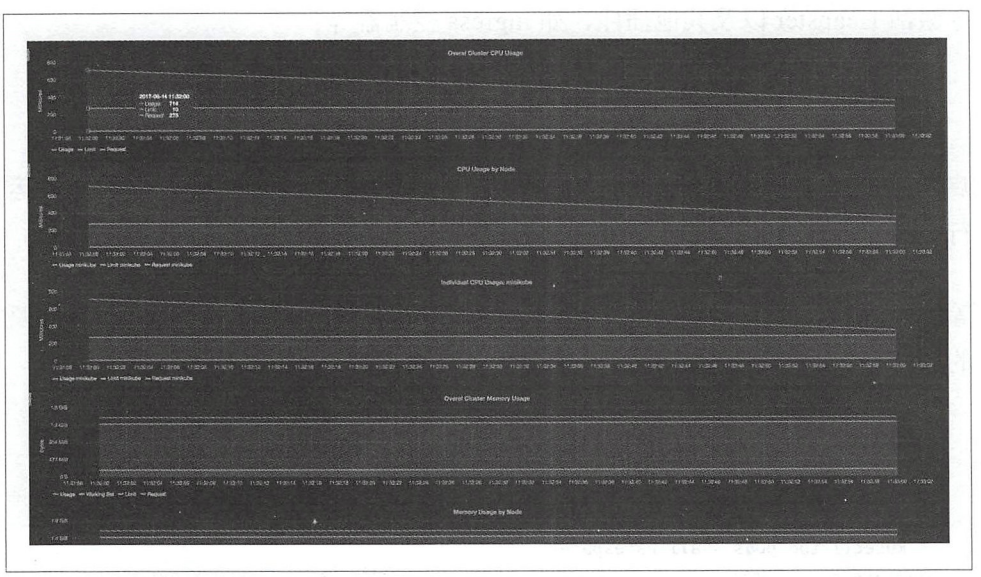


图 11-1：Grafana 仪表盘的截图，显示了 Minikube 的各项指标

请注意现在可以在 Grafana 上深入查看各项指标了。

11.6 在 Minikube 上使用 Prometheus

问题

如何以集中的方式查看集群的系统指标和应用程序指标?

解决方案

可以按照以下步骤使用 Prometheus：

1. 创建一个配置映射文件保存 Prometheus 的配置。
2. 为 Prometheus 设置服务账号，并通过 RBAC（请参阅 10.3 节）赋予该服务账号（请参阅 10.1 节）访问所有指标的权限。
3. 为 Prometheus 创建一个应用，它将包括一个部署、一个服务及一个 Ingress 资源，所以可以在集群外通过浏览器访问该应用。

首先，需要通过 ConfigMap 对象（请参阅 8.3 节关于配置映射的介绍）设置 Prometheus。后面的 Prometheus 应用会用到该对象。创建一个名为 *prometheus.yml* 文件保存 Prometheus 的配置，其内容如下所示：

```
global:
  scrape_interval:     5s
  evaluation_interval: 5s
scrape_configs:
- job_name:            'kubernetes-nodes'
  scheme:              https
  tls_config:
    ca_file:           /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    server_name:       'gke-k8scb-default-pool-be16f9ee-522p'
    insecure_skip_verify: true
  bearer_token_file:   /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role:              node
  relabel_configs:
```

```
    - action:                labelmap
      regex:                 __meta_kubernetes_node_label_(.+)
  - job_name:                'kubernetes-cadvisor'
    scheme:                  https
    tls_config:
      ca_file:               /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    bearer_token_file:       /var/run/secrets/kubernetes.io/serviceaccount/token
    kubernetes_sd_configs:
    - role:                  node
    relabel_configs:
    - action:                labelmap
      regex:                 __meta_kubernetes_node_label_(.+)
    - target_label:          __address__
      replacement:           kubernetes.default.svc:443
    - source_labels:         [__meta_kubernetes_node_name]
      regex:                 (.+)
      target_label:          __metrics_path__
      replacement:           /api/v1/nodes/${1}:4194/proxy/metrics
```

我们可以利用此文件创建配置映射，如下所示：

```
$ kubectl create configmap prom-config-cm --from-file=prometheus.yml
```

接下来，设置 Prometheus 的服务账号，并在名为 *prometheus-rbac.yaml* 的清单文件中指定角色（权限），如下所示：

```
apiVersion:  v1
kind:        ServiceAccount
metadata:
  name:      prometheus
  namespace: default
---
apiVersion:  rbac.authorization.k8s.io/v1beta1
kind:        ClusterRoleBinding
metadata:
  name:      prometheus
roleRef:
  apiGroup:  rbac.authorization.k8s.io
  kind:      ClusterRole
  name:      cluster-admin
subjects:
- kind:      ServiceAccount
  name:      prometheus
  namespace: default
```

现在可以使用该清单文件，创建服务账号，并指定角色：

```
$ kubectl create -f prometheus-rbac.yaml
```

现在所有的前提条件都已经准备就绪（配置和访问权限），可以开始配置 Prometheus 应用。前面说过，该应用包括一个部署、一个服务和一个 Ingress 资源，并且用到了上一个步骤中定义的配置映射和服务账号。

接下来，在 *prometheus-app.yaml* 清单文件中定义该 Prometheus 应用：

```
kind:                                   Deployment
apiVersion:                             extensions/v1beta1
metadata:
  name:                                 prom
  namespace:                            default
  labels:
    app:                                prom
spec:
  replicas:                             1
  selector:
    matchLabels:
      app:                              prom
  template:
    metadata:
      name:                             prom
      labels:
        app:                            prom
    spec:
      serviceAccount:                   prometheus
      containers:
      - name:                           prom
        image:                          prom/prometheus
        imagePullPolicy:                Always
        volumeMounts:
        - name:                         prometheus-volume-1
          mountPath:                    "/prometheus"
        - name:                         prom-config-volume
          mountPath:                    "/etc/prometheus/"
      volumes:
      - name:                           prometheus-volume-1
        emptyDir:                       {}
      - name:                           prom-config-volume
        configMap:
          name:                         prom-config-cm
          defaultMode:                  420
---
kind:                                   Service
apiVersion:                             v1
metadata:
  name:                                 prom-svc
  labels:
    app:                                prom
```

```
spec:
  ports:
  - port:                                    80
    targetPort:                              9090
  selector:
    app:                                     prom
  type:                                      LoadBalancer
  externalTrafficPolicy:                     Cluster
---
kind:                                        Ingress
apiVersion:                                  extensions/v1beta1
metadata:
  name:                                      prom-public
  annotations:
    ingress.kubernetes.io/rewrite-target:    /
spec:
  rules:
  - host:
    http:
      paths:
      - path:                                /
        backend:
          serviceName:                       prom-svc
          servicePort:                       80
```

现在利用清单文件创建该应用：

```
$ kubectl create -f prometheus-app.yaml
```

恭喜你创建了一个成熟的应用！现在用户可以通过 `$MINISHIFT_IP/graph`（例如：`https://192.168.99.100/graph`）路径访问 Prometheus，并可以看到如图 11-2 所示的内容。

讨论

Prometheus 是个强大且灵活的监视与预警系统。你可以利用它，最好是再加上某个测量代码库（*https://prometheus.io/docs/instrumenting/clientlibs/*）让应用汇报高层次的的指标，如汇报运行过的交易数量等，就像 kubelet 汇报 CPU 使用率一样。

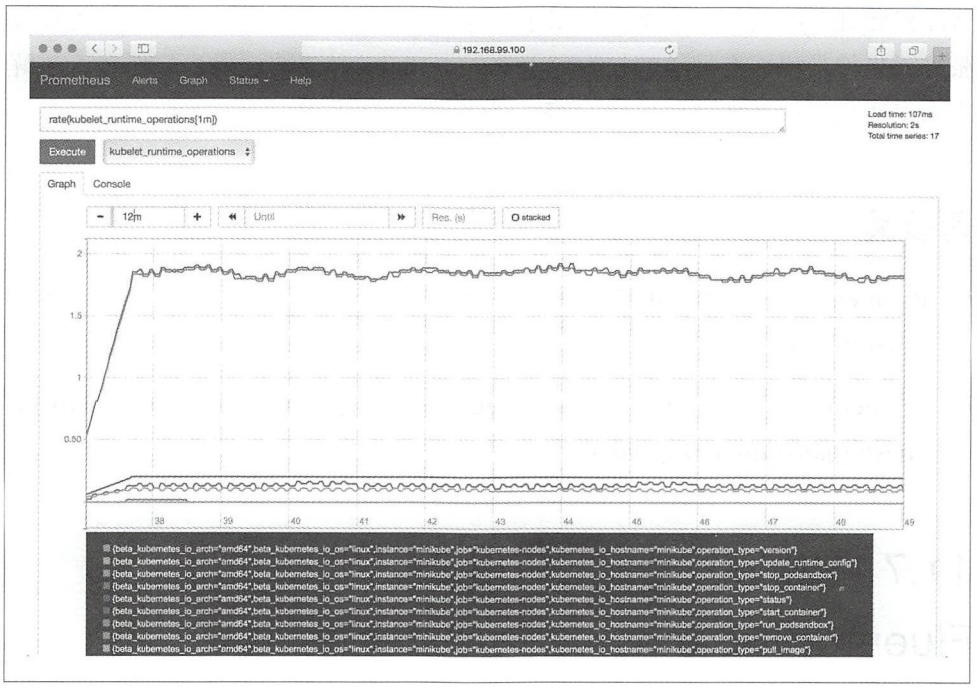


图 11-2：Prometheus 截图

Prometheus 速度快且具有扩展性，但是需要借助其他软件完成各项指标的可视化。典型的做法是与 Grafana 相连接。

在 Kubernetes 的 1.7.0 版本到 1.7.2 版本中使用 Prometheus 存在已知的问题，因为 `kubelet` 显示容器指标的行为在 v1.7.0 中发生了改变。

请注意解决方案中所介绍的 Prometheus 的配置在 v1.7.0 到 v1.7.2 中可以正常使用，但是如果在 v1.7.3 以及更新的版本中使用时，需要参考“Prometheus 配置文件实例”（*https://github.com/prometheus/prometheus/blob/master/documentation/examples/prometheus-kubernetes.yml#L88*）一文，进行相应的修改。

请注意此处介绍的解决方案不仅限于 Minikube。事实上，只要可以创建服务账号（也就是说，有足够的权限赋予 Prometheus 必要的权限），那么这个解决方案适用于很多环境，包括 GKE、ACS 或 OpenShift。

请参阅

- Prometheus 测量的相关文档（*https://prometheus.io/docs/practices/instrumentation/*）。
- Prometheus 中 Grafana 与 Prometheus 的使用文档（*https://prometheus.io/docs/visualization/grafana/*）。

11.7 在 Minikube 上使用 Elasticsearch–Fluentd–Kibana

问题

如何以集中的方式查看集群中所有应用的日志?

解决方案

可以使用 Elasticsearch，Fluentd 和 Kibana，具体内容如下。

首先，为了做好准备，请确保为 Minikube 分配了足够的资源。例如，使用 `--cpus=4 --memory=4000` ，并确保 ingress 插件已被激活，如下所示：

```
$ minikube start
Starting local Kubernetes v1.7.0 cluster...
Starting VM...
Getting VM IP address...
Moving files into cluster...
Setting up certs...
Starting cluster components...
Connecting to cluster...
```

```
Setting up kubeconfig...
Kubectl is now configured to use the cluster.

$ minikube addons list | grep ingress
- ingress: enabled
```

如果插件没有激活，那么可以通过如下命令激活：

```
$ minikube addons enable ingress
```

接下来，创建名为 *efk-logging.yaml* 的清单文件，具体内容如下：

```
kind:                                       Ingress
apiVersion:                                 extensions/v1beta1
metadata:
  name:                                     kibana-public
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host:
    http:
      paths:
      - path:                               /
        backend:
          serviceName:                      kibana
          servicePort:                      5601
---
kind:                                       Service
apiVersion:                                 v1
metadata:
  labels:
    app:                                    efk
  name:                                     kibana
spec:
  ports:
  - port:                                   5601
  selector:
    app:                                    efk
---
kind:                                       Deployment
apiVersion:                                 extensions/v1beta1
metadata:
  name:                                     kibana
spec:
  replicas:                                 1
  template:
    metadata:
      labels:
        app:                                efk
```

```
    spec:
      containers:
      - env:
        - name:                         ELASTICSEARCH_URL
          value:                        http://elasticsearch:9200
        name:                           kibana
        image:                          docker.elastic.co/kibana/kibana:5.5.1
        ports:
          - containerPort:              5601
---
kind:                                   Service
apiVersion:                             v1
metadata:
  labels:
    app:                                efk
  name:                                 elasticsearch
spec:
  ports:
  - port:                               9200
  selector:
    app:                                efk
---
kind:                                   Deployment
apiVersion:                             extensions/v1beta1
metadata:
  name:                                 es
spec:
  replicas:                             1
  template:
    metadata:
      labels:
        app:                            efk
    spec:
      containers:
      - name:                           es
        image:                          docker.elastic.co/elasticsearch/
                                        elasticsearch:5.5.1
        ports:
        - containerPort:                9200
        env:
        - name:                         ES_JAVA_OPTS
          value:                        "-Xms256m -Xmx256m"
---
kind:                                   DaemonSet
apiVersion:                             extensions/v1beta1
metadata:
  name:                                 fluentd
spec:
  template:
    metadata:
      labels:
        app:                            efk
      name:                             fluentd
```

```
      spec:
        containers:
        - name:                          fluentd
          image:                         gcr.io/google_containers/fluentd-
                                         elasticsearch:1.3
          env:
           - name:                       FLUENTD_ARGS
             value:                      -qq
          volumeMounts:
           - name:                       varlog
             mountPath:                  /varlog
           - name:                       containers
             mountPath:                  /var/lib/docker/containers
        volumes:
          - hostPath:
              path:                      /var/log
            name:                        varlog
          - hostPath:
              path:                      /var/lib/docker/containers
            name:                        containers
```

现在可以启动 EFK 栈了：

```
$ kubectl create -f efk-logging.yaml
```

所有应用都启动以后，可以利用下面的用户名和密码登录 Kibana：

- 用户名：kibana。
- 密码：changeme。

打开 https://$IP/app/kibana#/discover?_g=()，并单击 Discover 页签，就可以看到日志了。

可以使用下列命令清理或重启 EFK 栈：

```
$ kubectl delete deploy/es && \
  kubectl delete deploy/kibana && \
  kubectl delete svc/elasticsearch && \
  kubectl delete svc/kibana && \
  kubectl delete ingress/kibana-public && \
  kubectl delete daemonset/fluentd
```

讨论

通过 Logstash 也可以查看日志。我们在解决方案中选择使用 Fluentd，是因为它是云原生计算基金会（Cloud Native Computing Foundation，CNCF）的项目，且备受关注。

请注意启动 Kibana 需要花费一定的花时间，而且可能需要反复加载几次该网络应用才能完成配置。

请参阅

- Manoj Bhagwat 的博文“在 Kubernetes 上利用 Fluentd 和 ElasticSearch 集中管理 Docker 日志”（*https://medium.com/@manoj.bhagwat60/to-centralize-your-docker-logs-with-fluentd-and-elasticsearch-on-kubernetes-42d2ac0e8b6c*）。
- Kubernetes 基于 AWS 的 EFK 栈（*https://github.com/Skillshare/kubernetes-efk*）。
- Elk-kubernetes（*https://github.com/kayrus/elk-kubernetes*）。

第 12 章

维护与故障排除

在本章的各节中，我们将介绍应用级别和集群级别的维护。我们还将从多个角度介绍故障排除，包括调试 pod 和容器，测试服务的连接性，解读资源的状态，以及节点维护。最后我们还将介绍如何使用 etcd，即 Kubernetes 的控制层面的存储组件。本章主要面向集群管理员和应用开发人员。

12.1 启用 kubectl 的自动补齐

问题

记住 kubectl 完整的命令和参数是一件很难的事情，那么如何在输入命令时使用自动补齐功能？

解决方案

启用 kubectl 的自动补齐。

对于 Linux 和 *bash* shell，可以在当前 shell 中通过以下命令激活 kubectl 的自动补齐：

```
$ source <(kubectl completion bash)
```

对于其他操作系统和 shell，请参阅文档（*https://kubernetes.io/docs/tasks/tools/install-kubectl/#enabling-shell-autocompletion*）。

请参阅

- kubectl 概览（*https://kubernetes.io/docs/reference/kubectl/overview/*）。
- kubectl 参考手册（*https://kubernetes.io/docs/reference/kubectl/cheatsheet/*）。

12.2 删除服务上的 pod

问题

对于定义良好并由几个 pod 支持的服务（请参阅 5.1 节），如果其中一个 pod 出现了问题，如何将这个 pod 从访问点列表中移除，以便稍后再进行检查？

解决方案

可以使用 `--overwrite` 参数重新标记该 pod，通过该参数可以改写 pod 上 `run` 命令的标签。通过改写标签，可以确保该 pod 不会被服务选择器（请参阅 5.1 节）选中，并从访问点列表中移除该 pod。同时，负责监视 pod 的副本集会发现一个 pod 消失不见了，从而启动一个新的副本。

要完成上述操作，首先使用 `kubectl run`（请参阅 4.4 节）生成一个简单的部署：

```
$ kubectl run nginx --image nginx --replicas 4
```

当查看 pod 时，请注意观察字段 `run` 中显示的标签，你会看到有 4 个 pod 的值为 `nginx`（`run=nginx` 是 `kubectl run` 命令自动生成的标签）：

```
$ kubectl get pods -Lrun
NAME                     READY     STATUS              RESTARTS   AGE       RUN
nginx-d5dc44cf7-5g45r    1/1       Running             0          1h        nginx
nginx-d5dc44cf7-l429b    1/1       Running             0          1h        nginx
```

```
nginx-d5dc44cf7-pvrfh    1/1       Running             0          1h        nginx
nginx-d5dc44cf7-vm764    1/1       Running             0          1h        nginx
```

然后为这个部署建立一个服务并查看访问点（每个访问点对应于各自的 pod IP 地址）：

```
$ kubectl expose deployments nginx --port 80

$ kubectl get endpoints
NAME         ENDPOINTS                                                  AGE
nginx        172.17.0.11:80,172.17.0.14:80,172.17.0.3:80 + 1 more...    1h
```

接下来，可以通过一个简单的命令将第一个 pod 从服务列表中删除：

```
$ kubectl label pods nginx-d5dc44cf7-5g45r run=notworking --overwrite
```

为了找到 pod 的 IP 地址，你可以通过 JSON 格式查看 pod 清单文件并运行一个 JQuery 查询：

```
$ kubectl get pods nginx-d5dc44cf7-5g45r -o json | \
  jq -r .status.podIP172.17.0.3
```

如下所示，一个新的标记为 `run=nginx` 的 pod 出现在列表中，且 notworking 的 pod 依然存在，但是不会再出现在服务访问点列表中：

```
$ kubectl get pods -Lrun
NAME                     READY     STATUS        RESTARTS   AGE       RUN
nginx-d5dc44cf7-5g45r    1/1       Running       0          21h       notworking
nginx-d5dc44cf7-hztlw    1/1       Running       0          21s       nginx
nginx-d5dc44cf7-l429b    1/1       Running       0          5m        nginx
nginx-d5dc44cf7-pvrfh    1/1       Running       0          5m        nginx
nginx-d5dc44cf7-vm764    1/1       Running       0          5m        nginx

$ kubectl describe endpoints nginx
Name:         nginx
Namespace:    default
Labels:       run=nginx
Annotations:  <none>
Subsets:
  Addresses:          172.17.0.11,172.17.0.14,172.17.0.19,172.17.0.7
...
```

12.3 从集群外部访问集群 IP 的服务

问题

如果内部服务出现问题，如何在不公开到外部的情况下，测试它在本地能否工作正常？

解决方案

可以通过 `kubectl proxy` 为 Kubernetes 的 API 服务器建立代理。

假设你已经按照 12.2 节中描述的步骤创建了一个部署和服务，那么在查询该服务的时候，可以看到 `nginx` 服务：

```
$ kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
nginx        ClusterIP   10.109.24.56   <none>        80/TCP    22h
```

从 Kubernetes 集群外部无法访问该服务。但是可以在另外一个终端窗口运行代理，然后在 `localhost` 上访问该服务。

首先在另外一个终端窗口运行代理：

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

你可以通过 `--port` 参数在指定的端口上运行代理。

然后，在先前的终端窗口中使用浏览器或 `curl` 访问该服务开放的应用程序。请注意需要在访问该服务的路径中加入“`/proxy`”，否则只能得到代表该服务的 JSON 对象：

```
$ curl http://localhost:8001/api/v1/proxy/namespaces/default/services/nginx/
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!<title>
...
```

请注意你可以通过 `curl` 访问 `localhost` 的方式访问整个 Kubernetes 的 API。

12.4 掌握并解析资源的状态

问题

如何在脚本或另外一个自动化环境。例如：持续集成和持续交付（Continuous Integration/Continuous Delivery，CI/CD）管道中，根据资源（比如 pod）的状态进行相应的操作？

解决方案

可以使用 `kubectl get $KIND/$NAME -o json`，并使用下述的方法解析 JSON 输出结果。

如果你安装了 JSON 查询工具 `jq`，那么可以使用它解析资源的状态。假设有一个叫做 `jump` 的 pod，现在要了解该 pod 的服务质量（Quality of Service，QoS）[注1] 水平：

```
$ kubectl get po/jump -o json | jq --raw-output .status.qosClass
BestEffort
```

请注意此处 `jq` 的参数 `--raw-output` 显示的是原始数据，而 `.status.qosClass` 是用于匹配相应子字段的表达式。

注 1：请参阅文档：Medium, "What are Quality of Service (QoS) Classes in Kubernetes"（*https://medium.com/google-cloud/quality-of-service-class-qos-in-kuberntes-bb76a89eb2c6*）。

再举一个关于查询事件或状态变化的例子：

```
$ kubectl get po/jump -o json | jq .status.conditions
[
  {
    "lastProbeTime": null,
    "lastTransitionTime": "2017-08-28T08:06:19Z",
    "status": "True",
    "type": "Initialized"
  },
  {
    "lastProbeTime": null,
    "lastTransitionTime": "2017-08-31T08:21:29Z",
    "status": "True",
    "type": "Ready"
  },
  {
    "lastProbeTime": null,
    "lastTransitionTime": "2017-08-28T08:06:19Z",
    "status": "True",
    "type": "PodScheduled"
  }
]
```

当然，这些查询不仅限于 pod，你可以用这些技巧查询任何资源。例如，可以查询部署的改版信息：

```
$ kubectl get deploy/prom -o json | jq .metadata.annotations
{
  "deployment.kubernetes.io/revision": "1"
}
```

或者可以查询构成服务的所有访问点：

```
$ kubectl get ep/prom-svc -o json | jq '.subsets'
[
  {
    "addresses": [
      {
        "ip": "172.17.0.4",
        "nodeName": "minikube",
        "targetRef": {
          "kind": "Pod",
          "name": "prom-2436944326-pr60g",
          "namespace": "default",
          "resourceVersion": "686093",
          "uid": "eee59623-7f2f-11e7-b58a-080027390640"
        }
```

```
        }
      ],
      "ports": [
        {
          "port": 9090,
          "protocol": "TCP"
        }
      ]
    }
  ]
```

以上是 jq 的介绍，现在我们再来看看一个不需要外部工具的方法，即使用 Go 模板的自带功能。

Go 编程语言可以在名为 text/template 的包内定义模板，而这些模板可以用于任何类型的数据转换，且 kubectl 自带对这种模板的支持。例如，可以通过如下命令，查看当前命名空间中的所有容器映像：

```
$ kubectl get pods -o go-template \
          --template="{{range .items}}{{range .spec.containers}}{{.image}} \
          {{end}}{{end}}"
busybox prom/prometheus
```

请参阅

- jq 指南（*https://stedolan.github.io/jq/manual/*）。
- jq 练习环境，无需安装 jq 也可以练习查询（*https://jqplay.org/*）。
- Go 语言的包模板（*https://golang.org/pkg/text/template/*）。

12.5 调试 pod

问题

如果遇到 pod 无法按照预期启动，或在一段时间后发生故障的情况，应当如何处理？

解决方案

为了系统地发现并修复问题产生的原因，我们需要采用 OODA 循环流程：

1. 观察（Observe）。容器日志中有什么？发生了什么事件？网络连通性如何？
2. 调整（Orient）。制定一套合理的假设，尽可能大胆地设想，但不要急于下结论。
3. 决定（Decide）。选择其中一种假设。
4. 行动（Act）。测试选择的假设。如果得到证实，那么问题就解决了；否则，重回第一步。

让我们来看一个 pod 发生故障的实例。首先创建一个明文 *unhappy-pod.yaml* 的清单文件，内容如下：

```
apiVersion:       extensions/v1beta1
kind:             Deployment
metadata:
  name:           unhappy
spec:
  replicas:       1
  template:
    metadata:
      labels:
        app:      nevermind
    spec:
      containers:
      - name:     shell
        image:    busybox
        command:
        - "sh"
        - "-c"
        - "echo I will just print something here and then exit"
```

现在启动该部署，并看看创建的 pod，会发现结果不太顺利：

```
$ kubectl create -f unhappy-pod.yaml
deployment "unhappy" created

$ kubectl  get po
NAME                       READY     STATUS             RESTARTS   AGE
unhappy-3626010456-4j251   0/1       CrashLoopBackOff   1          7s
```

```
$ kubectl describe po/unhappy-3626010456-4j251
Name:           unhappy-3626010456-4j251
Namespace:      default
Node:           minikube/192.168.99.100
Start Time:     Sat, 12 Aug 2017 17:02:37 +0100
Labels:         app=nevermind
                pod-template-hash=3626010456
Annotations:    kubernetes.io/created-by={"kind":"SerializedReference","apiVersion":
"v1","reference":{"kind":"ReplicaSet","namespace":"default","name":
"unhappy-3626010456","uid":
"a9368a97-7f77-11e7-b58a-080027390640"...
Status:         Running
IP:             172.17.0.13
Created By:     ReplicaSet/unhappy-3626010456
Controlled By:  ReplicaSet/unhappy-3626010456
...
Conditions:
  Type          Status
  Initialized   True
  Ready         False
  PodScheduled  True
Volumes:
  default-token-rlm2s:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-rlm2s
    Optional:   false
QoS Class:      BestEffort
Node-Selectors: <none>
Tolerations:    <none>
Events:
  FirstSeen   ...   Reason                Message
  ---------   ...   ------                -------
  25s         ...   Scheduled             Successfully assigned
                                          unhappy-3626010456-4j251 to minikube
  25s         ...   SuccessfulMountVolume MountVolume.SetUp succeeded for
                                          volume "default-token-rlm2s"
  24s         ...   Pulling               pulling image "busybox"
  22s         ...   Pulled                Successfully pulled image "busybox"
  22s         ...   Created               Created container
  22s         ...   Started               Started container
  19s         ...   BackOff               Back-off restarting failed container
  19s         ...   FailedSync            Error syncing pod
```

如上所述，Kubernetes 认为该 pod 没有准备好服务访问，因为它遇到了一个“error syncing pod”的错误。

另一种查看上述信息的方式是使用 Kubernetes 的仪表盘，查看部署（见图 12-1）以及监控的副本集与 pod（见图 12-2）。

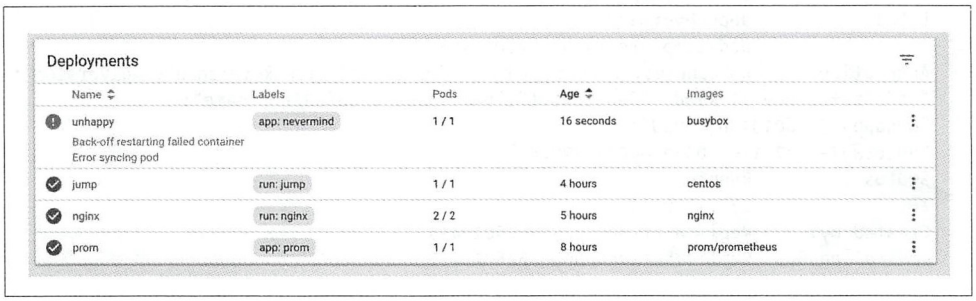


图 12-1：出错状态下的部署截图

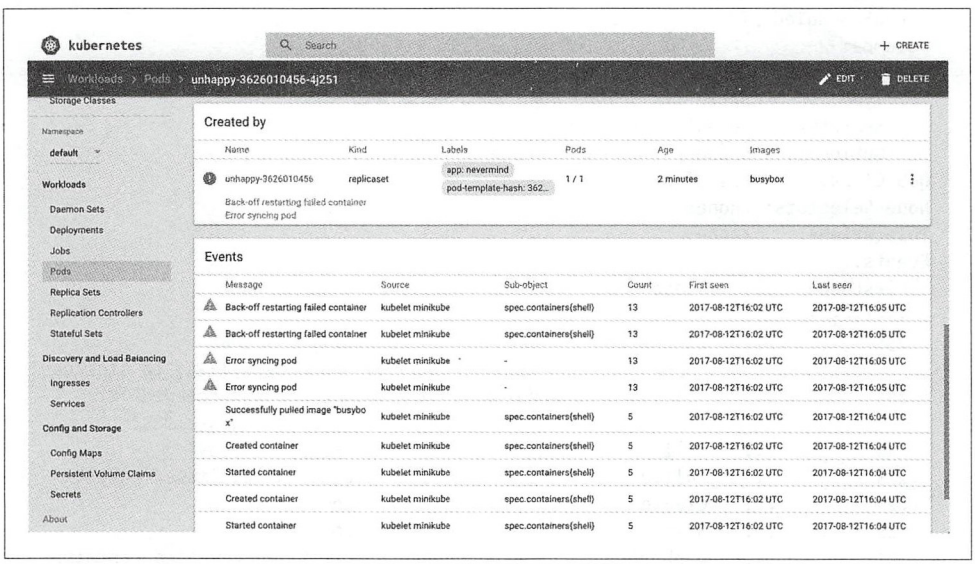


图 12-2：出错状态下 pod 的截图

讨论

导致 pod 故障或节点行为异常的原因可能不尽相同。在怀疑软件 bug 之前请先检查以下若干事项：

- 清单文件正确吗？请结合 Kubernetes 的 JSON 结构（*https://github.com/garethr/kubernetes-json-schema*）进行排查。
- 容器是独立运行的吗？是在本地（也就是在 Kubernetes 之外）运行的吗？
- Kubernetes 可以访问容器的注册，以及查看容器的映像吗？
- 节点之间可以互相对话吗？
- 节点可以访问主节点吗？
- 集群的 DNS 工作正常吗？
- 节点上有足够的资源吗？
- 容器的资源使用是否受限？

请参阅

- Kubernetes 故障排除应用程序文档（*https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application/*）。
- 应用程序的自我检查和调试（*https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application-introspection/*）。
- 调试 pod 和副本控制器（*https://kubernetes.io/docs/tasks/debug-application-cluster/debug-pod-replication-controller/*）。
- 调试服务（*https://kubernetes.io/docs/tasks/debug-application-cluster/debug-service/*）。
- 集群的故障排除（*https://kubernetes.io/docs/tasks/debug-application-cluster/debug-cluster/*）。

12.6 集群状态的详细快照

问题

如何才能获得整个集群状态的详细快照，以便于调整、检查或排除故障？

解决方案

使用`kubectl cluster-info dump`命令。例如，在子目录*cluster-state-2017-08-13*中创建集群状态的转储，如下所示：

```
$ kubectl cluster-info dump --all-namespaces \
  --output-directory=$PWD/cluster-state-2017-08-13

$ tree ./cluster-state-2017-08-13
.
├── default
│   ├── cockroachdb-0
│   │   └── logs.txt
│   ├── cockroachdb-1
│   │   └── logs.txt
│   ├── cockroachdb-2
│   │   └── logs.txt
│   ├── daemonsets.json
│   ├── deployments.json
│   ├── events.json
│   ├── jump-1247516000-sz87w
│   │   └── logs.txt
│   ├── nginx-4217019353-462mb
│   │   └── logs.txt
│   ├── nginx-4217019353-z3g8d
│   │   └── logs.txt
│   ├── pods.json
│   ├── prom-2436944326-pr60g
│   │   └── logs.txt
│   ├── replicasets.json
│   ├── replication-controllers.json
│   └── services.json
├── kube-public
│   ├── daemonsets.json
│   ├── deployments.json
│   ├── events.json
│   ├── pods.json
│   ├── replicasets.json
│   ├── replication-controllers.json
│   └── services.json
```

```
├── kube-system
│   ├── daemonsets.json
│   ├── default-http-backend-wdfwc
│   │   └── logs.txt
│   ├── deployments.json
│   ├── events.json
│   ├── kube-addon-manager-minikube
│   │   └── logs.txt
│   ├── kube-dns-910330662-dvr9f
│   │   └── logs.txt
│   ├── kubernetes-dashboard-5pqmk
│   │   └── logs.txt
│   ├── nginx-ingress-controller-d2f2z
│   │   └── logs.txt
│   ├── pods.json
│   ├── replicasets.json
│   ├── replication-controllers.json
│   └── services.json
└── nodes.json
```

12.7 添加 Kubernetes 工作节点

问题

如何向 Kubernetes 集群添加工作节点？

解决方案

根据环境的需要（例如，在裸金属裸机环境中需要在机架上安装一台新服务器，或在公有云环境中，需要建立新的虚拟机等）准备一台新的服务器，然后安装三个组件建立 Kubernetes 的工作节点：

`kubelet`

`kubelet` 是节点管理员，负责监控所有的 pod，无论是 API 服务器控制的 pod，还是在本地运行的静态 pod。

请注意 `kubelet` 是最后的仲裁者，可以决定哪些 pod 可以在节点上运行，哪些不可以，并负责：

- 向 API 服务器报告节点和 pod 的状态。
- 定期地执行存活探针。
- 挂载 pod 卷并下载 secret。
- 控制容器的运行时。

容器的运行时
: 负责下载容器映像以及运行容器。最初它是 Docker 引擎固有的功能，但是如今它是基于容器运行时接口（ Container Runtime Interface，CRI）的插件系统，所以可以使用 CRI-O 等代替 Docker。

`kube-proxy`
: 此进程可以根据节点的规则自动配置 iptables 规则，从而激活 Kubernetes 服务的抽象层（将 VIP 重新指向一个或多个代表该服务的 pod 访问点）。

这些组件的安装要根据实际的环境情况以及选用的安装方法（云端、`kubeadm` 等）决定。可选的参数列表，请参照 `kubelet` 参考手册（*https://kubernetes.io/docs/reference/generated/kubelet/*）以及 `kube-proxy` 参考手册（*https://kubernetes.io/docs/reference/generated/kube-proxy/*）。

讨论

工作节点与 Kubernetes 的部署或服务等其他资源不同，并非由 Kubernetes 管理中心直接创建，Kubernetes 只负责管理。这也就意味着，在 Kubernetes 中创建节点的时候，实际上仅创建了代表工作节点的对象。Kubernetes 根据节点的 `metadata.name` 字段进行健康检查，从而判断节点的合法性，如果节点合法（即所有必要的组件都处于运行状态），就将其归为集群的一部分；否则，所有的集群活动都会忽略此节点，直到节点合法。

请参阅

- Kubernetes 架构设计文档中关于“Kubernetes 节点”（*https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture/architecture.md#the-kubernetes-node*）。
- 主节点的交流（*https://kubernetes.io/docs/concepts/architecture/master-node-communication/*）。
- 静态 pod（*https://kubernetes.io/docs/tasks/administer-cluster/static-pod/*）。

12.8 抽出 Kubernetes 节点以实施维护

问题

如何对节点实施维护工作，例如打安全补丁或升级操作系统？

解决方案

可以使用 `kubectl drain` 命令。例如，如果要对节点 `123-worker` 实施维护：

```
$ kubectl drain 123-worker
```

在准备将节点放回服务的时候，可以使用 `kubectl uncordon 123worker` 命令，节点就可以被调度了。

讨论

`kubectl drain` 命令所做的工作包括：首先将指定的节点标记为不可调度，从而阻止新 pod 被分配到节点上（实质上是 `kubectl cordon`）。然后如果 API 服务器支持则收回所有 pod。否则就使用常见的 `kubectl delete` 命令删除 pod。Kubernetes 的文档详细描绘了这些步骤的时序图，如图 12-3 所示。

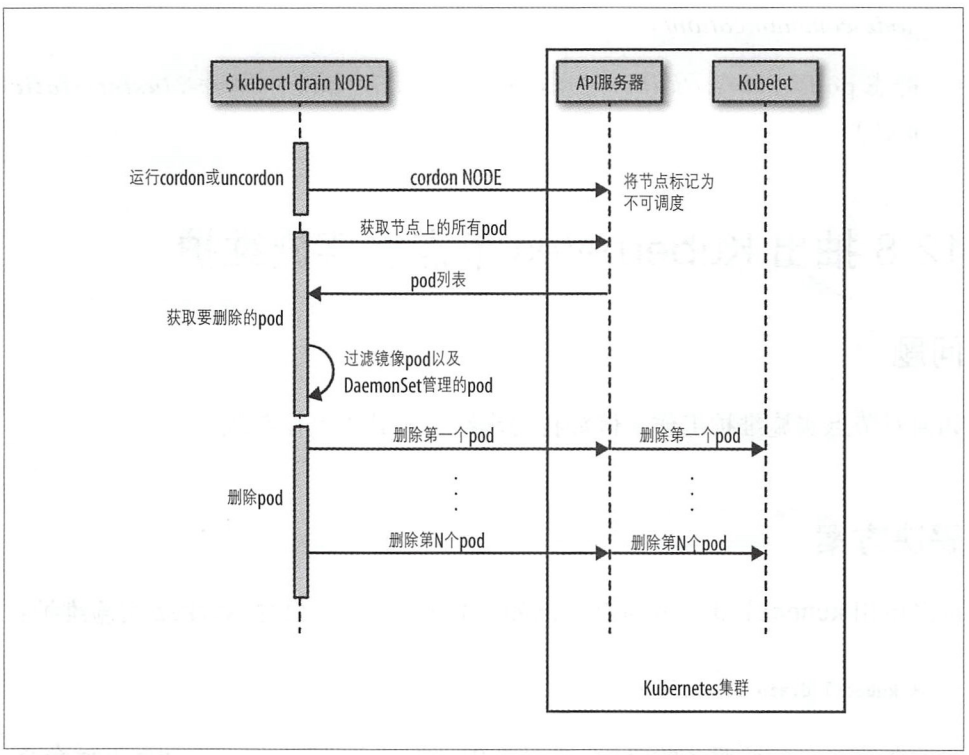


图 12-3：抽出节点的序列图

`kubectl drain` 命令会回收或删除所有的 pod，除了镜像 pod（API 服务器无法删除镜像 pod）。对于由 `DaemonSet` 监管的 pod，如果不使用 `--ignore-daemonsets`，`drain` 命令则无法执行，且无论怎样它都不会删除任何由 `DaemonSet` 管理的 pod（这些 pod 会被 `DaemonSet` 控制器立即替换，所以会忽略不可调度的标记）。

drain 命令会等待正常的结束，所以在 kubectl drain 命令结束前不应该操作该节点。请注意 kubectl drain $NODE --force 也可以收回 pod，除非 pod 由 RC、RS、工作、DaemonSet 或 StatefulSet 管理。

请参阅

- 在考虑应用程序 SLO 的同时安全的抽出节点（*https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/*）。
- kubectl 的参考文档（*https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#drain*）。

12.9 管理 etcd

问题

如何访问 etcd 以进行备份，或直接检验集群的状态？

解决方案

可以使用 curl 或 etcdctl 访问并查询 etcd。例如，在 Minikube 中（假设安装了 jq）：

```
$ minikube ssh

$ curl 127.0.0.1:2379/v2/keys/registry | jq .
{
  "action": "get",
  "node": {
    "key": "/registry",
    "dir": true,
    "nodes": [
      {
```

```
        "key": "/registry/persistentvolumeclaims",
        "dir": true,
        "modifiedIndex": 241330,
        "createdIndex": 241330
      },
      {
        "key": "/registry/apiextensions.k8s.io",
        "dir": true,
        "modifiedIndex": 641,
        "createdIndex": 641
      },
...
```

这个方法适用于任何加载了 etcd v2 API 的环境。

讨论

etcd 是 Kubernetes 控制中心的组件。API 服务器（请参阅 6.1 节）是无状态的，且是唯一可以直接与 etcd 交流的 Kubernetes 组件，etcd 是管理集群的状态的分布式存储组件。本质上，etcd 是以键值的形式保存的，在 etcd2 中键组成了层次结构，但是根据 etcd3 的介绍，存储方式改为了扁平化结构（并向后兼容层次结构的键）。

一直到 Kubernetes 1.5.2，我们都在使用 etcd2，从那以后我们换成了 etcd3。在 Kubernetes 1.5.x 中，etcd3 还在使用 v2 的 API 模式，之后会换成 etcd v3 API，而 v2 很快就会被淘汰。尽管从开发人员的角度看来这些变化并没什么影响，因为 API 服务器会处理好交互的抽象，但是作为管理员需要注意 API 使用的 etcd 版本。

一般来讲，理应由集群管理员负责管理 etcd，即负责升级并备份数据。在特定环境中，如果由控制中心管理 etcd，比如 Google Kubernetes 引擎，那就无法直接访问 etcd。这是设计上的考虑，没有其他方法可以解决这个问题。

请参阅

- Etcd v2 集群管理员指南（*https://coreos.com/etcd/docs/latest/v2/admin_guide.html*）。
- Etcd v3 灾难恢复指南（*https://coreos.com/etcd/docs/latest/op-guide/recovery.html*）。
- 操控 Kubernetes 的 etcd 集群（*https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/*）。
- MInikube 文档中“从 pod 内部访问 Localkube 资源：etcd 实例”（*https://github.com/kubernetes/minikube/blob/master/docs/accessing_etcd.md*）。
- Stefan Schimanski 和 Michael Hausenblas 的博文“深度探索 Kubernetes：API 服务器之二”（*https://blog.openshift.com/kubernetes-deep-dive-api-server-part-2/*）。
- Michael Hausenblas 的博文“从 etcd2 转换到 etcd3 的注意点”（*https://hackernoon.com/notes-on-moving-from-etcd2-to-etcd3-dedb26057b90*）。

第 13 章

Kubernetes 开发

到本章为止，我们学习了如何安装 Kubernetes，如何与 Kubernetes 交互，以及使用 Kubernetes 部署和管理应用程序，本章中我们将介绍如何根据需求改进 Kubernetes，以及如何修改 Kubernetes 的 bug。为此，我们需要安装 Go，并需要从 GitHub 上访问 Kubernetes 的源代码。我们将介绍如何编译整个 Kubernetes，以及如何编译客户端 `kubectl` 等具体的组件。我们还将演示如何使用 Python 调用 Kubernetes 的 API 服务，以及如何使用自定义的资源扩展 Kubernetes。

13.1 编译源代码

问题

如何从源代码制作自己的 Kubernetes 可执行文件，而不是下载官方发行的可执行文件（请参阅 2.4 节），或第三方的产品？

解决方案

克隆 Kubernetes Git 代码仓库，并编译源代码。

如果在 Docker 的主机上，可以使用顶层 *Makefile* 的 `quick-release` 目标，如下所示：

```
$ git clone https://github.com/kubernetes/kubernetes
$ cd kubernetes
$ make quick-release
```

基于 Docker 的编译需要至少 4GB 的内存。请确保 Docker 服务可以使用的内存足够。对于 MacOS，可以通过 Docker for Mac 选项增加分配给 Docker 的内存。

编译好的可执行文件保存在 *_output/release-stage* 目录中，完整的包在 *_output/release-tars* 目录中。

如果有设置好的 Golang 环境，那么可以使用 *Makefile* 的 `release` 目标：

```
$ git clone https://github.com/kubernetes/Kubernetes
$ cd kubernetes
$ make
```

编译好的可执行文件保存在 *_output/bin* 目录中。

请参阅

- 具体的 Kubernetes 开发者指南（*https://github.com/kubernetes/community/tree/master/contributors/devel*）。

13.2 编译特定的组件

问题

如何从源代码编译特定 Kubernetes 组件，而不是所有的组件，例如，编译客户端 `kubectl`？

解决方案

这里我们不使用 13.1 节中介绍的 `make quick-release` 或 `make`，而是使用 `make kubectl`。

顶层 *Makefile* 中包含用于编译各个组件的目标。例如，编译 `kubectl`、`kubeadm` 和 `hyperkube` 的命令如下所示：

```
$ make kubectl
$ make kubeadm
$ make hyperkube
```

编译好的可执行文件保存在 *_output/bin* 目录中。

13.3 如何使用 Python 客户端与 Kubernetes API 交互

问题

如何在 Python 中使用 Kubernetes API 编写脚本？

解决方案

首先需要安装 Python 的 `kubernetes` 模块。这个模块是在 Kubernetes incubator 中开发的。可以从源代码或从 Python 软件包管理工具（Python Package Index，PyPi）网站安装此模块：

```
$ pip install kubernetes
```

通过默认的 `kubectl` 环境可以访问 Kubernetes 的集群，然后就可以使用该 Python 模块与 Kubernetes API 对话了。例如，如下 Python 脚本可以查看所有的 pod 并输出它们的名称：

```
from kubernetes import client, config

config.load_kube_config()

v1 = client.CoreV1Api()
res = v1.list_pod_for_all_namespaces(watch=False)
for pod in res.items:
    print(pod.metadata.name)
```

脚本中 `config.load_kube_config()` 的调用，可以从 `kubectl` 的配置文件中加载 Kubernetes 的认证信息和访问点。默认情况下，它将加载当前环境中的集群访问点和认证信息。

讨论

Python 客户端是通过 Kubernetes API 的 OpenAPI 规格创建的。它会自动生成并一直保持最新。该客户端可以访问所有的 API。

每个 API 组对应一个具体的类，所以如果需要调用属于 `/api/v1` API 组的 API 对象的方法，那么需要实例化 `CoreV1Api` 类。在使用部署时，需要实例化 `extensionsV1beta1Api` 类。你可以在自动生成的帮助文档中找到所有的方法与对应的 API 组的实例（*https://github.com/kubernetes-client/python/tree/master/kubernetes*）。

请参阅

- 项目代码仓库中的实例（*https://github.com/kubernetes-client/python/tree/master/examples*）。

13.4 使用自定义的资源扩展 API

问题

假设你有一个自定义的工作负荷，但没有任何已有的资源（比如，部署、job

或 StatefulSet）适合描述这个工作负荷。如何利用代表该工作负荷的新资源扩展 Kubernetes API，且可以继续像往常一样使用 kubectl？

解决方案

可以使用自定义资源定义（Custom Resource Definition，CRD）。

假设需要定义一个 Function 类型的自定义资源。它代表了短期运行的、像 Job 一样的资源，类似于 AWS Lambda 提供的功能，也就是所谓的“功能即服务”（Function-as-a-Service，FaaS，它还有个极具误导性的名字“无服务器体系”）。

关于在 Kubernetes 上运行适合生产环境的 FaaS 解决方案，请参阅 14.7 节。

首先，在一个名为 *functions-crd.yaml* 的清单文件中定义该 CRD：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind:       CustomResourceDefinition
metadata:
  name:     function.example.com
spec:
  group:    example.com
  version:  v1
  names:
    kind:   Function
    plural: functions
    scope:  Namespaced
```

然后，通知 API 服务器这个新的 CRD（注册需要几分钟的时间）：

```
$ kubectl create -f functions-crd.yaml
customresourcedefinition "functions.example.com" created
```

现在自定义的资源 Function 已经定义好了，并且服务器也知道了它的存在，接下来可以使用一个名为 *myfaas.yaml* 的清单文件实例化这个资源，如下所示：

```
apiVersion: example.com/v1
kind:       Function
```

```
metadata:
  name:     myfaas
spec:
  code:     "http://src.example.com/myfaas.js"
  ram:      100Mi
```

照例还需创建 Function 类型的 myfaas 的资源：

```
$ kubectl create -f myfaas.yaml
function "myfaas" created

$ kubectl get crd functions.example.com -o yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  creationTimestamp: 2017-08-13T10:11:50Z
  name: functions.example.com
  resourceVersion: "458065"
  selfLink: /apis/apiextensions.k8s.io/v1beta1/customresourcedefinitions
            /functions.example.com
  uid: 278016fe-81a2-11e7-b58a-080027390640
spec:
  group: example.com
  names:
    kind: Function
    listKind: FunctionList
    plural: functions
    singular: function
  scope: Namespaced
  version: v1
status:
  acceptedNames:
    kind: Function
    listKind: FunctionList
    plural: functions
    singular: function
  conditions:
  - lastTransitionTime: null
    message: no conflicts found
    reason: NoConflicts
    status: "True"
    type: NamesAccepted
  - lastTransitionTime: 2017-08-13T10:11:50Z
    message: the initial names have been accepted
    reason: InitialNamesAccepted
    status: "True"
    type: Established

$ kubectl describe functions.example.com/myfaas
Name:           myfaas
Namespace:      default
```

```
Labels:         <none>
Annotations:    <none>
API Version:    example.com/v1
Kind:           Function
Metadata:
  Cluster Name:
  Creation Timestamp:                  2017-08-13T10:12:07Z
  Deletion Grace Period Seconds:       <nil>
  Deletion Timestamp:                  <nil>
  Resource Version:                    458086
  Self Link:                           /apis/example.com/v1/namespaces/default
                                       /functions/myfaas
  UID:                                 316f3e99-81a2-11e7-b58a-080027390640
Spec:
  Code: http://src.example.com/myfaas.js
  Ram:  100Mi
Events: <none>
```

现在只需通过访问API服务器就可以找到CRD。例如，使用`kubectl proxy`可以从本地访问API服务器，还可以查询键空间（比如下例中查询了`example.com/v1`）：

```
$ curl 127.0.0.1:8001/apis/example.com/v1/ | jq .
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "example.com/v1",
  "resources": [
    {
      "name": "functions",
      "singularName": "function",
      "namespaced": true,
      "kind": "Function",
      "verbs": [
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "create",
        "update",
        "watch"
      ]
    }
  ]
}
```

这里可以看到资源列表以及允许的操作。

如果想要删除自定义的资源实例 myfaas，那么只需运行删除命令：

```
$ kubectl delete functions.example.com/myfaas
function "myfaas" deleted
```

讨论

正如你所见，创建一个 CRD 很直观。从终端用户的角度看来，CRD 代表一个稳定的 API，几乎与 pod 或 job 等内置的资源没有什么区别。所有常见的命令，比如 `kubectl get` 和 `kubectl delete`，也可以照常工作。

然而，创建一个 CRD 其实只是完全扩展该 Kubernetes API 所必需的工作的一小半。CRD 仅仅实现了通过 API 服务在 etcd 中存储数据或获取数据。还需要编写自定义的控制器，用于解释自定义数据表达用户的意图和建立控制循环对比当前的状态和声明的状态，以及调和二者。

在 v1.7 之前，CRD 被称作第三方资源（*third-party resources*，TPRs）。如果你手头有一个 TPR，强烈建议现在就移植它。

CRD 主要的局限性在于（因此在某些场合，你可能希望使用用户的 API 服务器）：

- 每个 CRD 仅支持一个版本，尽管每个 API 组可以有多个版本（这意味着不可以在不同的 CRD 代表之间互相转换）。
- 在 v1.7 或更早的版本中，CRD 不支持给字段赋默认值。
- 从 v1.8 开始，支持在 CRD 规格中使用字段验证规则。
- 不可以定义子资源，比如 status 资源。

请参阅

- 使用 CRD 扩展 Kubernetes API（*https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-custom-resource-definitions/*）。
- Stefan Schimanski 和 Michael Hausenblas 的博文“深入 Kubernetes：API 服务器之三”（*https://blog.openshift.com/kubernetes-deep-dive-api-server-part-3a/*）。
- Aaron Levy 在 KubeCon 2017 大会上的演讲“编写自定的控制器：扩展集群的功能”（*https://www.youtube.com/watch?v=_BuqPMlXfpE*）。
- Tu Nguyen 的文章“深入 Kubernetes 控制器”（*https://engineering.bitnami.com/articles/a-deep-dive-into-kubernetes-controllers.html*）。
- Yaron Haviv 的文章“使用自定义资源扩展 Kubernetes1.7”（*https://thenewstack.io/extend-kubernetes-1-7-custom-resources/*）。

第 14 章

Kubernetes 的生态系统

本章中，我们将介绍更广泛的 Kubernetes 生态系统，即 Kubernetes incubator 的软件以及相关的项目，例如，Helm 和 kompose。

14.1 安装 Helm（Kubernetes 的包管理器）

问题

如果不想手写所有的 Kubernetes 清单文件，如何通过命令行从仓库中查找一个包，然后下载并安装？

解决方案

可以使用 Helm。Helm 是 Kubernetes 包管理器；它将 Kubernetes 包定义为一系列清单文件和一些元数据。这些清单文件其实就是模板。Helm 实例化包的时候，会给模板中的字段赋值。Helm 包被称作图表。

Helm 包括客户端命令行界面 `helm` 和服务器端 `tiller`。可以使用 `helm` 与图表交互，而 `tiller` 可以作为普通的 Kubernetes 部署在 Kubernetes 集群内运行。

你可以从源代码编译 Helm，或从 GitHub 的发行页面下载（地址是：*https://github.com/kubernetes/helm/releases*），解压归档文件，并将 `helm` 可执行

文件放入 $PATH 目录。例如，在 MacOS 上，可以通过如下步骤安装 Heml v2.7.2：

```
$ wget https://storage.googleapis.com/kubernetes-helm/ \
  helm-v2.7.2-darwin-amd64.tar.gz

$ tar -xvf helm-v2.7.2-darwin-amd64.tar.gz


$ sudo mv darwin-amd64/64 /usr/local/bin

$ helm version
```

现在 helm 命令已经加入了 $PATH，可以利用它在 Kubernetes 集群上启动服务端组件 tiller。下面使用 Minikube 为例：

```
$ kubectl get nodes
NAME       STATUS    AGE       VERSION
minikube   Ready     4m        v1.7.8

$ helm init
$HELM_HOME has been configured at /Users/sebgoa/.helm.

Tiller (the helm server side component) has been installed into your Kubernetes
Cluster. Happy Helming!

$ kubectl get pods --all-namespaces | grep tiller
kube-system   tiller-deploy-1491950541-4kqxx   0/1  ContainerCreating  0  1s
```

现在都设置好了，你可以开始从 100 多个包（*https://hub.kubeapps.com/*）中挑选需要的进行安装。

14.2 利用 Helm 安装应用程序

问题

如果已经安装了 helm 命令（请参阅 14.1），如何查找并部署图表呢？

解决方案

默认情况下，Helm 自带一些配置好的图表仓库。这些仓库是由社区维护的，

详情请参阅 GitHub（*https://github.com/kubernetes/charts*）。其上有 100 多个可利用的图表。

例如，假设你想部署 Redis。那么可以在 Helm 仓库中查找 `redis` 并安装。Helm 会获取该图表并创建一个实例，该实例称为发行。

首先，确认 `tiller` 处于运行状态，且默认的仓库配置完成：

```
$ kubectl get pods --all-namespaces | grep tiller
kube-system   tiller-deploy-1491950541-4kqxx   1/1   Running   0   3m

$ helm repo list
NAME    URL
stable  http://storage.googleapis.com/kubernetes-charts
```

现在查找 Redis 包：

```
$ helm search redis
NAME                    VERSION       DESCRIPTION
stable/redis            0.5.1    Open source, advanced key-value store. It ...
testing/redis-cluster   0.0.5    Highly available Redis cluster with multiple...
testing/redis-standalone 0.0.1   Standalone Redis Master
stable/sensu            0.1.2    Sensu monitoring framework backed by the ...
testing/example-todo    0.0.6    Example Todo application backed by Redis
```

并使用 `helm install` 创建发行，如下所示：

```
$ helm install stable/redis
```

Helm 将创建该图表中定义的所有 Kubernetes 对象；例如，一个 secret（请参阅 8.2 节），一个 PVC（请参阅 8.5 节），一个服务（请参阅 5.1 节），以及一个部署。所有这些对象组成了一个 Helm 发行，可以作为一个单元管理。

最终将得到一个能够正常运行的 `redis` pod：

```
$ helm ls
NAME           REVISION        UPDATED                   STATUS          CHART
...
broken-badger  1         Fri May 12 11:50:43 2017  DEPLOYED  redis-0.5.1  ...

$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
broken-badger-redis-4040604371-tcn14   1/1       Running   0          3m
```

更多关于 Helm 图表，以及如何创建自己的图表，请参阅 14.3 节。

14.3 利用 Helm 创建自己的图表打包应用程序

问题

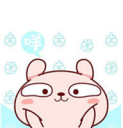
现有一个包含多个 Kubernetes 清单文件的应用程序，如何将它打包成 Helm 图表？

解决方案

可以使用 `helm create` 和 `helm package` 命令。

通过 `helm create`，可以生成图表的骨架。在终端窗口中执行该命令，指定图表的名称。例如，创建一个名为 `oreilly` 的图表：

```
$ helm create oreilly
Creating oreilly


$ tree oreilly/
oreilly/
├── Chart.yaml
├── charts
├── templates
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── ingress.yaml
│   └── service.yaml
└── values.yaml

2 directories, 7 files
```

如果你已经写好了所有的清单文件，那么可以将它们复制到 */templates* 目录下并删除骨架中生成的清单文件。如果想把你的清单文件变成模板，那么可以将清单文件中需要替换的部分写入 *values.yaml* 中。接下来编辑 *Chart.yaml* 文件中的元数据，并将所有独立的图表放入 */charts* 目录中。

你可以通过以下命令在本地测试图表：

```
$ helm install ./oreilly
```

最后使用 `helm package oreilly/` 命令打包。该命令将生成图表的 tarball，将它复制到本地的图表代码库中，并为本地代码库生成一个新的 *index.yaml* 文件。查看 *~/.helm*，应该能看到如下类似的内容：

```
$ ls -l ~/.helm/repository/local/
total 16
-rw-r--r--  1 sebgoa  staff   379 Dec 16 21:25 index.yaml
-rw-r--r--  1 sebgoa  staff  1321 Dec 16 21:25 oreilly-0.1.0.tgz
```

现在 `helm search oreilly` 命令应该返回本地的图表：

```
$ helm search oreilly
NAME            VERSION  DESCRIPTION
local/oreilly0.1.0       A Helm chart for Kubernetes
```

请参阅

- Kubernetes 的 Bitnami 文档中关于“如何创建第一个 Helm 图表”（*https://docs.bitnami.com/kubernetes/how-to/create-your-first-helm-chart/*）。
- Helm 文档关于“图表最佳实践指南”（*https://docs.helm.sh/chart_best_practices/*）。

14.4 将 Docker Compose 文件转换成 Kubernetes 清单文件

问题

假设你通过 Docker 访问容器，并编写了 Docker compose 文件来定义多个容器的应用程序。现在如何使用 Kubernetes，以及如何重用 Docker compose 文件？

解决方案

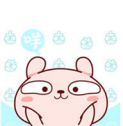
可以使用 `kompose`。这是一个命令行工具，可以将 Docker compose 文件转换成 Kubernetes 清单文件。

首先，从 GitHub 上下载 `kompose`，为了方便起见加入到 `$PATH` 中。例如，在 MacOS 上可以运行如下命令：

```
$ wget https://github.com/kubernetes-incubator/kompose/releases/download/ \
      v1.6.0/kompose-darwin-amd64

$ sudo mv kompose-darwin-amd64 /usr/local/bin/kompose

$ sudo chmod +x /usr/local/bin/kompose

$ kompose version
1.6.0 (ae4ef9e)
```

如果有下列用于启动 `redis` 容器的 Docker compose 文件：

```
version: '2'

services:
  redis:
    image: redis
    ports:
    - "6379:6379"
```



那么可以通过如下命令，将它自动转换成 Kubernetes 清单文件：

```
$ kompose convert --stdout
```

清单文件将显示在 stdout，其中可以看到一个 Kubernetes 服务和一个部署。想要自动创建这些对象，可以使用 up 命令，如下所示：

```
$ kompose up
```

一些 Docker compose 指令不会被转换成 Kubernetes。这种情况下，kompose 将输出警告信息，通知你转换未成功。

一般来说不会有大问题，但是转换生成的清单文件可能无法在 Kubernetes 正常工作。这类的转换一般都不会太完美。然而，这些转换可以帮助你创建基本的 Kubernetes 清单文件。最值得注意的是，卷和网络隔离的处理一般都需要手动定制。

讨论

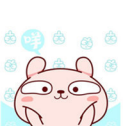
kompose 的主要命令包括 convert、up 和 down。请在命令行界面内使用 --help 参数查看每个命令的具体帮助信息。

默认情况下，kompose 会将 Docker 服务转换成 Kubernetes 部署和相关的服务。你也可以指定使用 DaemonSet（请参阅 7.3 节），或者使用 OpenShift 的专用对象，例如 DeploymentConfiguration。

14.5 使用 kubicorn 创建 Kubernetes 集群

问题

如何在 AWS 上创建 Kubernetes 集群？



解决方案

可以使用 kubicorn 在 AWS 上创建 Kubernetes 集群。目前 kubicorn 没有可以执行文件的形式发布，所以下面的操作需要先安装 Go 才能进行。

首先，安装 kubicorn，并确认安装了 Go（1.8 或更高版本）。在这里，我们使用 CentOS 环境。

```
$ go version
go version go1.8 linux/amd64

$ yum group install "Development Tools" \
  yum install ncurses-devel

$ go get github.com/kris-nova/kubicorn
...
Create, Manage, Image, and Scale Kubernetes infrastructure in the cloud.

Usage:
  kubicorn [flags]
  kubicorn [command]


Available Commands:
  adopt       Adopt a Kubernetes cluster into a Kubicorn state store
  apply       Apply a cluster resource to a cloud
  completion  Generate completion code for bash and zsh shells.
  create      Create a Kubicorn API model from a profile
  delete      Delete a Kubernetes cluster
  getconfig   Manage Kubernetes configuration
  help        Help about any command
  image       Take an image of a Kubernetes cluster
  list        List available states
  version     Verify Kubicorn version

Flags:
  -C, --color         Toggle colorized logs (default true)
  -f, --fab           Toggle colorized logs
  -h, --help          help for kubicorn
  -v, --verbose int   Log level (default 3)

Use "kubicorn [command] --help" for more information about a command.
```

安装了 kubicorn 命之令后，可以通过选择 *profile* 创建集群资源，并确认正确地定义了资源：

```
$ kubicorn create --name k8scb --profile aws
2017-08-14T05:18:24Z [✔]  Selected [fs] state store
```


```
2017-08-14T05:18:24Z [✿]  The state [./_state/k8scb/cluster.yaml] has been...

$ cat _state/k8scb/cluster.yaml
SSH:
  Identifier: ""
  metadata:
    creationTimestamp: null
  publicKeyPath: ~/.ssh/id_rsa.pub
  user: ubuntu
cloud: amazon
kubernetesAPI:
  metadata:
    creationTimestamp: null
  port: "443"
location: us-west-2
...
```

默认的资源 profile 假设在 *~/.ssh* 中有一对名叫 `id_rsa` 的私钥和 `id_rsa.pub` 的公钥。如果公钥和私钥不在 *~/.ssh* 下，那么可能需要做些改动。请注意以上我们使用的区域是 Oregon，`us-west-2`。

接下来，你需要准备一个 AWS 身份和访问管理用户（Identity and Access Management，IAM），并拥有如下权限：`AmazonEC2FullAccess`、`AutoScalingFullAccess` 和 `AmazonVPCFullAccess`。如果你没有这样的 IAM 用户，现在就创建一个吧[注 1]。

kubicorn 的最后一项准备工作是将 IAM 的认证设置到环境变量中，如下所示：

```
$ export AWS_ACCESS_KEY_ID=**************************
$ export AWS_SECRET_ACCESS_KEY=****************************************
```

现在根据上述资源的定义以及提供的 AWS 访问权限创建集群：

```
$ kubicorn apply --name k8scb
2017-08-14T05:45:04Z [✔]  Selected [fs] state store
2017-08-14T05:45:04Z [✔]  Loaded cluster: k8scb
2017-08-14T05:45:04Z [✔]  Init Cluster
2017-08-14T05:45:04Z [✔]  Query existing resources
2017-08-14T05:45:04Z [✔]  Resolving expected resources
```

注 1：请参阅文档：AWS Identity and Access Management User Guide，“Creating an IAM User in Your AWS Account”（*https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html*）。

```
2017-08-14T05:45:04Z [✔]  Reconciling
2017-08-14T05:45:07Z [✔]  Created KeyPair [k8scb]
2017-08-14T05:45:08Z [✔]  Created VPC [vpc-7116a317]
2017-08-14T05:45:09Z [✔]  Created Internet Gateway [igw-e88c148f]
2017-08-14T05:45:09Z [✔]  Attaching Internet Gateway [igw-e88c148f] to VPC ...
2017-08-14T05:45:10Z [✔]  Created Security Group [sg-11dba36b]
2017-08-14T05:45:11Z [✔]  Created Subnet [subnet-50c0d919]
2017-08-14T05:45:11Z [✔]  Created Route Table [rtb-8fd9dae9]
2017-08-14T05:45:11Z [✔]  Mapping route table [rtb-8fd9dae9] to internet gate...
2017-08-14T05:45:12Z [✔]  Associated route table [rtb-8fd9dae9] to subnet ...
2017-08-14T05:45:15Z [✔]  Created Launch Configuration [k8scb.master]
2017-08-14T05:45:16Z [✔]  Created Asg [k8scb.master]
2017-08-14T05:45:16Z [✔]  Created Security Group [sg-e8dca492]
2017-08-14T05:45:17Z [✔]  Created Subnet [subnet-cccfd685]
2017-08-14T05:45:17Z [✔]  Created Route Table [rtb-76dcdf10]
2017-08-14T05:45:18Z [✔]  Mapping route table [rtb-76dcdf10] to internet gate...
2017-08-14T05:45:19Z [✔]  Associated route table [rtb-76dcdf10] to subnet ...
2017-08-14T05:45:54Z [✔]  Found public IP for master: [34.213.102.27]
2017-08-14T05:45:58Z [✔]  Created Launch Configuration [k8scb.node]
2017-08-14T05:45:58Z [✔]  Created Asg [k8scb.node]
2017-08-14T05:45:59Z [✔]  Updating state store for cluster [k8scb]
2017-08-14T05:47:13Z [✿]  Wrote kubeconfig to [/root/.kube/config]
2017-08-14T05:47:14Z [✿]  The [k8scb] cluster has applied successfully!
2017-08-14T05:47:14Z [✿]  You can now `kubectl get nodes`
2017-08-14T05:47:14Z [✿]  You can SSH into your cluster ssh -i ~/.ssh/id_rsa ...
```

尽管书上看不到漂亮的颜色，但是实际上最后四行是绿色的，告诉你一切设置成功。你也可以在浏览器中访问亚马逊的 EC2 控制台来进行确认，如图 14-1 所示。

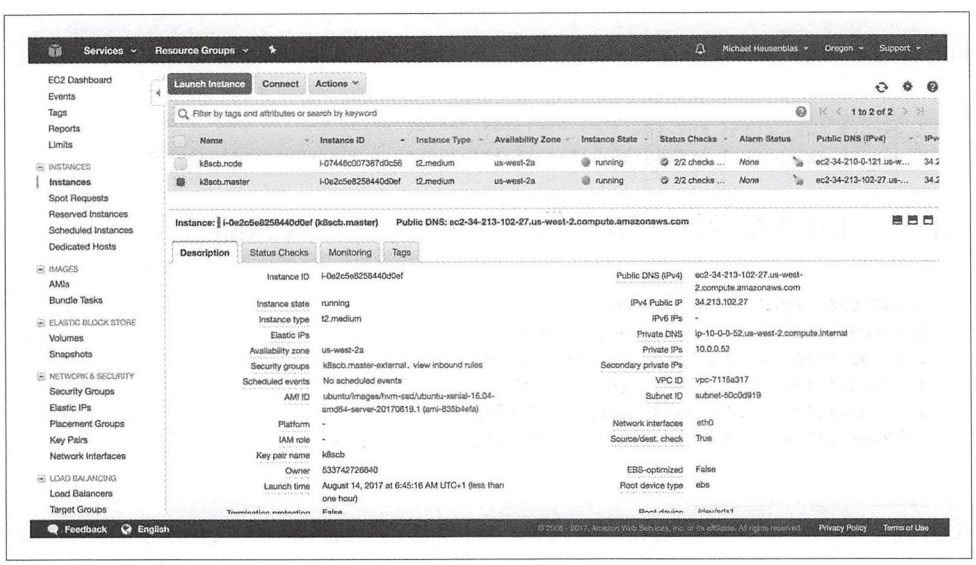


图 14-1：亚马逊 EC2 控制台截图，显示了两个由 kubicorn 创建的节点

现在，按照 kubicorn apply 命令最后一行输出的指示，在集群中运行 ssh：

```
$ ssh -i ~/.ssh/id_rsa ubuntu@34.213.102.27
The authenticity of host '34.213.102.27 (34.213.102.27)' can't be established.
ECDSA key fingerprint is ed:89:6b:86:d9:f0:2e:3e:50:2a:d4:09:62:f6:70:bc.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '34.213.102.27' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-1020-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

75 packages can be updated.
32 updates are security updates.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-10-0-0-52:~$ kubectl get all -n kube-system
NAME                                          READY     STATUS
po/calico-etcd-qr3f1                          1/1       Running
po/calico-node-9t472                          2/2       Running
po/calico-node-qlpp6                          2/2       Running
po/calico-policy-controller-1727037546-f152z  1/1       Running
po/etcd-ip-10-0-0-52                          1/1       Running
po/kube-apiserver-ip-10-0-0-52                1/1       Running
po/kube-controller-manager-ip-10-0-0-52       1/1       Running
po/kube-dns-2425271678-zcfdd                  0/3       ContainerCreating
po/kube-proxy-3s2c0                           1/1       Running
po/kube-proxy-t10ck                           1/1       Running
po/kube-scheduler-ip-10-0-0-52                1/1       Running

NAME              CLUSTER-IP      EXTERNAL-IP   PORT(S)         AGE
svc/calico-etcd   10.96.232.136   <none>        6666/TCP        4m
svc/kube-dns      10.96.0.10      <none>        53/UDP,53/TCP   4m

NAME                              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/calico-policy-controller   1         1         1            1           4m
deploy/kube-dns                   1         1         1            0           4m

NAME                                        DESIRED   CURRENT   READY     AGE
rs/calico-policy-controller-1727037546      1         1         1         4m
rs/kube-dns-2425271678                      1         1         0         4m
```

完成上述工作之后，别忘了销毁 Kubernetes 集群了（请注意可能需要花费几分钟的时间）：

```
$ kubicorn delete --name k8scb
2017-08-14T05:53:38Z [✔]  Selected [fs] state store
Destroying resources for cluster [k8scb]:
2017-08-14T05:53:41Z [✔]  Deleted ASG [k8scb.node]
...
2017-08-14T05:55:42Z [✔]  Deleted VPC [vpc-7116a317]
```

讨论

尽管 kubicorn 还是一个很新的项目，但是它已经发展健全，你还可以在 Azure 和 Digital Ocean 上使用它创建集群。

因为 kubicorn 没有发布可执行文件，所以你需要安装 Go，但是 kubicorn 在配置方面非常灵活，操作非常直观，尤其是如果你有管理员的背景。

请参阅

- kubicorn 文档“在 AWS 中设置 Kubernetes”（*http://kubicorn.io/documentation/aws-walkthrough.html*）。
- Lachlan Evenson 的视频介绍“使用 kubicorn 在 Digital Ocean 上建立 Kubernetes 集群”（*https://www.youtube.com/watch?v=XpxgSZ3dspE*）。

14.6 在版本控制中保存加密的 secret

问题

如何将所有 Kubernetes 的清单文件都保存在版本控制中并安全地分享（甚至公开），包括 secret？

解决方案

可以使用 sealed-secrets。Sealed-secrets 是一个 Kubernetes 的控制器，它可以解密单向加密的 secret，并在集群内创建 Secret 对象（请参阅 8.2 节）。

敏感的信息可以在 SealedSecret 对象中加密编码，SealedSecret 对象是一个自定义的 CRD 资源（请参阅 13.4 节）。SealedSecret 可以安全地保存到版本控制中，并可以公开分享。一旦在 Kubernetes 的 API 服务器上创建 SealedSecret，这个控制器就会解密并创建相应的 Secret 对象（该对象仅进行 base64 编码）。

首先，下载最新版本的 kubeseal 可执行文件，用于加密 secret：

```
$ GOOS=$(go env GOOS)

$ GOARCH=$(go env GOARCH)

$ wget https://github.com/bitnami/sealed-secrets/releases/download/v0.5.1/
      kubeseal-$GOOS-$GOARCH

$ sudo install -m 755 kubeseal-$GOOS-$GOARCH /usr/local/bin/kubeseal
```

然后创建 SealedSecret CRD，并启动该控制器：

```
$ kubectl create -f https://github.com/bitnami/sealed-secrets/releases/
                    download/v0.5.1/sealedsecret-crd.yaml

$ kubectl create -f https://github.com/bitnami/sealed-secrets/releases/
                    download/v0.5.1/controller.yaml
```

然后，你就可以得到一个新的自定义资源，和一个运行在 kube-system 命名空间中的新 pod：

```
$ kubectl get customresourcedefinitions
NAME                          AGE
sealedsecrets.bitnami.com     34s

$ kubectl get pods -n kube-system | grep sealed
sealed-secrets-controller-867944df58-l74nk   1/1       Running   0          38s
```

现在可以使用 sealed-secrets 了。首先，创建一个通用的 secret 清单文件：

```
$ kubectl create secret generic oreilly --from-literal=password=root -o json
                                        --dry-run > secret.json

$ cat secret.json
{
    "kind": "Secret",
```

```
    "apiVersion": "v1",
    "metadata": {
        "name": "oreilly",
        "creationTimestamp": null
    },
    "data": {
        "password": "cm9vdA=="
    }
}
```

创建清单文件的时候，可以使用 `--dry-run` 参数以避免在 API 服务器上创建对象。上述命令可以将清单文件输出到 `stdout`。如果想创建 YAML 格式的文件，可以使用 `-o yaml` 参数；如果想创建 JSON 格式的文件，则可以使用 `-o json`。

接下来使用 kubeseal 命令创建新的自定义 SealedSecret 对象：

```
$ kubeseal < secret.json > sealedsecret.json

$ cat sealedsecret.json
{
  "kind": "SealedSecret",
  "apiVersion": "bitnami.com/v1alpha1",
  "metadata": {
    "name": "oreilly",
    "namespace": "default",
    "creationTimestamp": null
  },
  "spec": {
    "data": "AgDXiFGoV6NKF8e9k1NeBMc5t4QmfZh3QKuDORAsFNCt5OwTwRhRLRAQOnzOsDk..."
  }
}
```

现在可以安全地将 *sealedsecret.json* 保存到版本控制中了。只有保存在 sealed-secret 控制中的私钥可以解密。在创建 SealedSecret 对象的时候，控制器会检测、解密并创建相应的加密信息：

```
$ kubectl create -f sealedsecret.json
sealedsecret "oreilly" created

$ kubectl get sealedsecret
NAME      AGE
oreilly   5s
```

```
$ kubectl get secrets
NAME       TYPE     DATA      AGE
...
oreilly    Opaque   1         5s
```

请参阅

- Sealed-secret 代码库（*https://github.com/bitnami-labs/sealed-secrets*）。
- Angus Lees 的文章“Sealed Secrets：保护 Kubernetes 的密码”（*https://engineering.bitnami.com/articles/sealed-secrets.html*）。

14.7 利用 kubeless 部署函数

问题

如何在不建立 Docker 容器的前提下，将 Python、Node.js、Ruby 或 PowerShell 的函数部署到 Kubernetes？以及如何通过 HTTP 或向消息总线发送事件的方式调用这些函数？

解决方案

可以使用 Kubernetes 本地的无服务器解决方案 kubeless。

kubeless 使用 CRD（请参阅 13.4 节）来定义 Function 对象，以及一个在 Kubernetes 集群内的 pod 中部署这些函数的控制器。

尽管此应用的范围非常广，但在本节中我们仅介绍部署一个 Python 函数（该函数可以返回发送的 JSON 格式数据）的基本例子。

首先，创建一个 kubeless 命名空间并启动控制器。可以从 GitHub 页面上获取 kubeless 每个版本发行的清单文件。从这个页面上还可以下载 kubeless 的可执行文件：

```
$ kubectl create ns kubeless

$ curl -sL https://github.com/kubeless/kubeless/releases/download/v0.3.1/ \
          kubeless-rbac-v0.3.1.yaml | kubectl create -f -

$ wget https://github.com/kubeless/kubeless/releases/download/v0.3.1/ \
       kubeless_darwin-amd64.zip

$ sudo cp bundles/kubeless_darwin-amd64/kubeless /usr/local/bin
```

kubeless 命名空间内有 3 个 pod：负责监视 Function 这个自定义访问点的控制器、Kafka pod 和 Zookeeper pod。

那些由事件触发的函数需要后两个 pod。对于由 HTTP 触发的函数，只需要让控制器处于运行状态：

```
$ kubectl get pods -n kubeless
NAME                                  READY     STATUS    RESTARTS   AGE
kafka-0                               1/1       Running   0          6m
kubeless-controller-9bff848c4-gnl7d   1/1       Running   0          6m
zoo-0                                 1/1       Running   0          6m
```

为了使用 kubeless，需要在名为 *post.py* 的文件中编写下列 Python 函数：

```
def handler(context):
    print context.json
    return context.json
```

然后使用 kubeless 命令行工具，在 Kubernetes 中部署该函数。function deploy 命令接受的可选参数包括：--runtime 参数用于指定编写函数的语言；--http-trigger 参数用于指定该函数可以被 HTTP(s) 调用触发；--handler 参数用于指定函数的名字，函数名的前缀则是存储该函数的文件名（去掉扩展名后的部分）；--from-file 参数指定了该函数所在的文件：

```
$ kubeless function deploy post-python --trigger-http \
                                       --runtime python2.7 \
                                       --handler post.handler \
                                       --from-file post.py
INFO[0000] Deploying function...
INFO[0000] Function post-python submitted for deployment
INFO[0000] Check the deployment status executing 'kubeless function ls post-
python'
```


```
$ kubeless function ls
NAME                NAMESPACE HANDLER       RUNTIME       TYPE     TOPIC
post-python   default  hellowithdata.handler     python      2.7      HTTP

$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
post-python-5bcb9f7d86-d7nbt   1/1     Running   0          6s
```

kubeless 控制器检测到新的 Function 对象，并创建了一个部署。函数的代码保存在配置映射中（请参阅 8.3 节），并在运行时添加到了运行中的 pod 中。然后就可以通过 HTTP 调用该函数了。如下是几个这类的对象：

```
$ kubectl get functions
NAME          AGE
post-python   2m

$ kubectl get cm
NAME          DATA      AGE
post-python   3         2m


$ kubectl get deployments
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
post-python   1         1         1            1           2m
```

可以使用 kubeless function call 命令调用该函数，如下所示：

```
$ kubeless function call post-python --data '{"oreilly":"function"}'
{"oreilly": "function"}
```

kubeless 的使用远远不止基本的由 HTTP 触发的函数。请通过 --help 参数在命令行界面内查看详情：kubeless --help。

请参阅

- kubeless 代码库（*https://github.com/kubeless/kubeless*）。
- kubeless 实例（*https://github.com/kubeless/kubeless/tree/master/examples*）。
- 在 Azure 容器服务上使用 kubeless（*https://github.com/kubeless/kubeless/blob/master/docs/kubeless-on-azure-container-services.md*）。

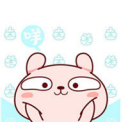
资源

常见资源

- Kubernetes 官方文档（*https://kubernetes.io/docs/home/*）。
- GitHub 上的 Kubernetes 代码库（*https://github.com/kubernetes/kubernetes/*）。
- GitHub 上的 Kubernetes 社区（*https://github.com/kubernetes/community/*）。

教程与实例

- Kubernetes 实例（*http://kubernetesbyexample.com/*）。
- Katacoda Kubernetes 练习环境（*https://www.katacoda.com/courses/kubernetes/playground*）。
- Brendan Burns、Kelsey Hightower 和 Joe Beda 的著作：《Kubernetes：Up and Running》（O'Reilly 出版）（*http://shop.oreilly.com/product/0636920043874.do*）。



作者介绍

Sébastien Goasguen 于 20 世纪 90 年代后期建立了第一个计算集群，他很自豪完成了博士学位，并非常感谢 Fortran 77 和偏微分方程。并行计算机带来的困难让他致力于使计算成为实用的工具，后来他还专注于网格和云计算。十五年以后，他心底十分希望容器和 Kubernetes 可以让他重回编写应用程序的工作。

目前他在 Bitnami 担任云技术高级总监，负责 Kubernetes 的工作。他于 2015 年底成立了 Kubernetes 创业公司 Skippbox。在 Skippbox 期间，他创建了多个开源软件应用程序和工具，用于增强 Kubernetes 的用户体验。他是 Apache 软件基金会的成员，也是 Apache CloudStack 的前副总裁。Sébastien 致力于云生态系统，并为几十个开源项目做出了贡献。他撰写了 Docker Cookbook，还是一位狂热的博客作者，并担任 Kubernetes 概念 Safari 订阅者的在线讲师。

Michael Hausenblas 是 Go、Kubernetes 和 Red Hat 的 OpenShift 的先驱开发者，他帮助 AppOps 构建和运行分布式服务。他有大规模数据处理和容器编排的背景，他在 W3C 和 IETF 的倡导和标准化方面有着丰富的经验。在 Red Hat 之前，Michael 曾在 Mesosphere、MapR 以及爱尔兰和奥地利的两个研究机构工作。他贡献开源软件（主要是使用 Go），博客，并经常活跃在 Twitter 上。

封面介绍

本书的封面动物是一只孟加拉鹰鸮（岩雕鸮，Bubo bengalensis）。这些大角猫头鹰经常成对出现，你可以在南亚的丘陵和岩石丛林中找到它们的身影。

孟加拉鹰鸮身高 19~22 英寸，重量在 39~70 盎司之间。它的羽毛呈棕灰色或米色，耳朵有棕色的毛簇。与其身体的中性色相比，它的眼睛颜色显橙色。这种橙色眼睛的猫头鹰在白天狩猎。它喜欢肉食，主要以老鼠或田鼠等啮齿类动物为食物，但也会在冬季期间吃其他鸟类。我们可以在黄昏和黎明时，听到这种猫头鹰发出的低沉而洪亮的“呜呼”的叫声。

雌鸟会在地面的浅凹处，岩壁和河岸筑巢，并产 2~5 颗奶油色的卵。33 天后鸟蛋孵化。雏鸟大约会在 10 周的时候长成大鸟，但是身体还没有成熟，在近 6 个月内仍然需要依靠父母。为了分散捕食者对后代的袭击，父母会假装翅膀受伤或以曲折的方式飞行。

O’Reilly 出版的图书，封面上很多动物都濒临灭绝。这些动物都是地球的至宝。如果你想知道如何保护这些动物，请访问 animals.oreilly.com。

封面图片来自 Meyers Kleines Lexicon。

O'REILLY®

Kubernetes经典实例

如果你们公司正在准备采用云端原生计算机架构，那么这本书将向你介绍如何成功地使用Kubernetes，它是自动部署、扩大与缩小规模以及容器化应用程序管理方面切实的标准。本书通过80多个久经考验的技巧，快速地向开发者、系统管理员和架构师介绍Kubernetes的入门知识，并掌握它所提供的强大的API。

在本书中，作者提供了在开发环境和产品环境中安装、使用Kubernetes以及与之交互的具体解决方案。并介绍了如何改造系统来满足具体的需求，以及熟悉Kubernetes更广泛的生态环境。每个章节介绍的技巧都以常用的“问题-解决方案-讨论”的过程来描述。

本书中的主要内容有：

- 创建Kubernetes集群。
- 使用Kubernetes命令行界面。
- 管理基本的workload类型。
- 使用服务。
- 探索Kubernetes API。
- 管理有状态的非云端原生应用。
- 使用卷与配置数据。
- 集群级别与应用程序级别的规模伸缩。
- 应用程序的安全。
- 监视与日志。
- 维护与排除故障。

“Kubernetes是史上最好的基础设施。这本书可以帮助你学习这个最好的基础设施。本书提供了能够解决实际问题的具体示例，你可以将其应用到实际工作中。认真阅读本书，你能够学习真正的技术，将自己的Kubernetes提升到一个新的水平。”

——Joe Beda
Heptio的CTO兼创始人，
Kubernetes创始人

Sébastien Goasguen是拥有20年开源经验的资深专家，而且还是Kubernetes早期的代码贡献者。他创建了Skippbox，这家创业公司开发了kompose和Cabin等Kubernetes工具。他目前在Bitnami担任高级云科技总监。

Michael Hausenblas是Go、Kubernetes和Red Hat的OpenShift的先驱开发者，他曾在Red Hat帮助AppOps建立和运营分发服务。在Red Hat之前，Michael从业于Mesosphere、MapR，并在爱尔兰和澳大利亚的研究机构担任要职。

O'Reilly Media, Inc.授权中国电力出版社出版

ISBN 978-7-5198-2399-3
9 787519 823993
定价：48.00元